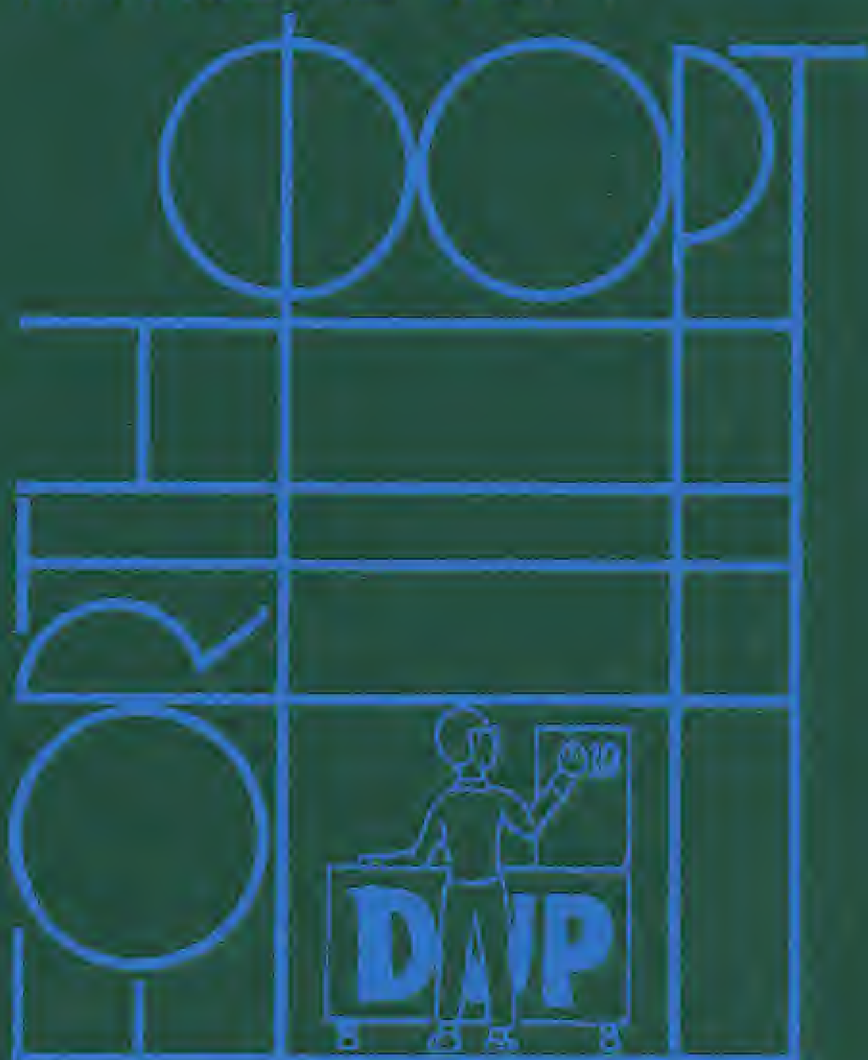
Л. Броуди

НАЧАЛЬНЫЙ КУРС ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ФОРТ

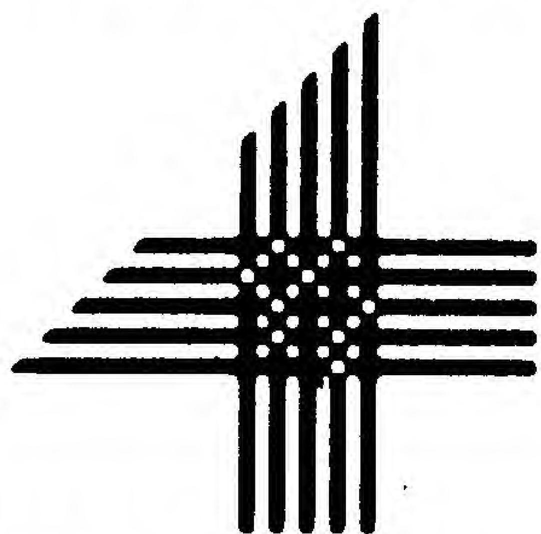


НАЧАЛЬНЫЙ КУРС ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ФОРТ

STARTING FORTH

AN INTRODUCTION TO THE **FORTH** LANGUAGE AND OPERATING SYSTEM FOR BEGINNERS AND PROFESSIONALS

LEO BRODIE
and
FORTH, INC.



With a foreword by
Charles H. Moore

PRENTICE-HALL, INC.
Englewood Cliffs, New Jersey 07632

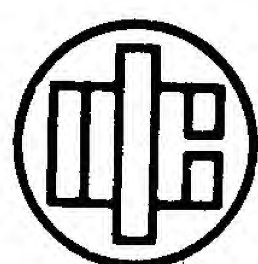
Л. Броуди

НАЧАЛЬНЫЙ КУРС ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ФОРТ

Перевод с английского В.А. Кондратенко

Под редакцией Б.А. Кацева, В.А. Кириллина

Предисловие И.В. Романовского



МОСКВА
"ФИНАНСЫ И СТАТИСТИКА"
1990

ББК 24.4.1
Б80

Броуди Л.

Б80 Начальный курс программирования на языке Форт: Пер. с англ.; Предисл. И. В. Романовского. — М.: Финансы и статистика, 1990. — 352 с.: ил.
ISBN 5-279-00252-6.

Книга является практическим руководством по программированию на приобретающем все бо́льшую популярность языке Форт. Рассматриваются основные принципы организации системы Форт. Приведено много примеров и наглядных иллюстраций, облегчающих восприятие излагаемого материала.
Для широкого круга специалистов.

$Б\frac{2404010000 - 014}{010(01) - 90}$ 129 — 89 ББК 24.4.1

ISBN 5-279-00252-6 (СССР)
ISBN 0-13-843079-9 (Великобритания)

© 1987 by Prentice-Hall, Inc.
© Перевод на русский язык, предисловие, примечания, «Финансы и статистика», 1990

К СОВЕТСКОМУ ЧИТАТЕЛЮ

С большим удовольствием приглашаю советских читателей в мир Форта. В списке моих адресатов есть представители всех стран, и только СССР до последнего времени составлял непонятное мне исключение.

Форт начинен идеями, а идеи легко преодолевают идеологические и национальные границы. К тому же Форт — это язык, и поскольку в его основе лежат слова, он легко переводим на естественные языки. Я уже видел французскую и китайскую версии Форта, где английские слова заменены соответствующими французскими и китайскими словами, так что программисты практически могут работать на своем родном языке. Я полагаю, то же самое может быть сделано и в вашей стране. Единственный недостаток подобной трансляции — уменьшается мобильность программ.

Компактный и эффективный язык программирования Форт является машинно-независимым. Моя десятилетняя практика реализации Форта на множестве универсальных, мини- и микрокомпьютеров показывает, что программы на всех ЭВМ выполняются до такой степени одинаково, что я забываю, на какой из машин работаю в данный момент. Чтобы перейти на новый компьютер, требуется всего несколько человеко-недель.

В основе гибкости Форта лежит его *простота.* Мне часто кажется, что люди слишком все усложняют. Некоторые языки программирования и операционные системы чересчур изощренны. Форт предлагает в качестве альтернативы простые средства программирования, которые можно расширять для обслуживания любой предметной области. Лео Броуди мастерски демонстрирует такие средства, подчеркивая их простоту.

Применяя Форт, пожалуйста, помните, что если вы делаете задачи сложнее, чем они есть на самом деле, то ничего в результате не выигрываете. Эффективная реализация простых алгоритмов, простота пользовательского интерфейса и представление простых

предсказуемых результатов — только это имеет значение. Изучите Форт — и ваш компьютер поможет вам полностью решить все свои проблемы.

Чак Мур,
Вудсайд СА
Сентябрь 1988

Я глубоко тронут тем, что моя книга издается в Советском Союзе. Хотя она уже распространена в англоязычных странах и переведена на немецкий, французский, японский, китайский и датский языки, данный перевод имеет для меня особое значение, потому что способствует обмену идеями между нашими двумя странами. В 50-х — 60-х годах, еще учась в школе, я даже представить себе не мог, что когда-нибудь напишу книгу, которая будет переведена на русский язык, и ее увидят советские читатели.

Все народы мира должны стремиться совместно использовать идеи и ресурсы, если мы хотим сохранить свою планету. Я горжусь тем, что эта книга — мой маленький вклад в наше общее дело.

Надеюсь, вы по достоинству оцените машинный язык, который создан господином Муром, и, на мой взгляд, является гениальной разработкой.

Лео Броуди

ПРЕДИСЛОВИЕ К РУССКОМУ ИЗДАНИЮ

В этой книге оказалось несколько хорошо написанных предисловий, и для традиционных слов советского специалиста пришлось искать подходящее место. Мне хочется вспомнить, как сам учился Форту, что оказалось неожиданно трудным делом. Было сделано несколько неудачных попыток, после чего один студент-первокурсник убедительно и просто продемонстрировал мне достоинства Форт-системы, отталкиваясь от которых я затем уже довольно легко справился со всем остальным.

Позднее я понял, что при первых попытках мне мешал предыдущий опыт: Форт слишком отличается от языков, обычно используемых в практическом программировании, — Алгола, Фортрана, Бейсика, Паскаля и т. п. Но именно отличия и делают Форт настолько интересным для программистов со стажем (конечно же, для тех из них, кто хочет расширить свои знания, что при достаточно большом опыте бывает не всегда).

Книга Л. Броуди в этом отношении построена очень хорошо. Приемы «занимательной игры» в ней как бы блокируют наш предшествующий серьезный опыт и первоначально «вводят в нас» систему нужных понятий как нечто новое, вне зависимости от приобретенных ранее навыков. Однако занимательность не делает материал менее серьезным — книга даст вам полную и детальную информацию обо всех важнейших конструкциях языка. Кстати, следует отметить, что вашему вниманию предлагается второе издание книги, исправленное и дополненное в соответствии со Стандартом-83. Первое издание, ориентированное на пользователей систем фиг[1]-Форта, вероятно, известно специалистам по оригиналу, а также по довольно широко распространенному переводу, выполненному Всесоюзным центром переводов. Когда вы прочтете достаточно много и Форт вам понравится, у вас возникнут естест-

[1] FIG-FORTH Interested Group.

венные вопросы: «А как у нас? Есть ли в Советском Союзе подобные системы, развивались ли такие идеи? Используется ли Форт в нашей стране и каковы перспективы его дальнейшего применения?»

Прежде всего напомним читателю, что язык, аналогичный Форту, предлагался и в Советском Союзе. В Московском университете, о чем упоминается в книге Н. П. Брусенцова «Микрокомпьютеры» (М.: Наука, 1985), в экспериментальной ЭВМ «Сетунь-70» были реализованы подобные технические идеи (двухстековой организации). Здесь же, в МГУ, на базе этих идей и уже с учетом Форта разработана диалоговая система структурного программирования.

В американском варианте, ставшем международным, Форт начал развиваться у нас в 80-е годы, главным образом в Ленинграде. Работы велись в разных местах почти одновременно, в том числе на математико-механическом факультете Ленинградского университета. Первоначально усилия были направлены на создание инструментальных средств для всех выпускаемых в стране процессоров (ЕС, СМ, К580, К1810, БЭСМ-6, «Эльбрус»). Описания отдельных групп и систем можно найти в книге С. Н. Баранова и Н. Р. Ноздрунова «Язык Форт и его реализации» (М.: Машиностроение, 1988). Недавно значительная часть ленинградской группы объединилась в научно-производственный кооператив «Форт-Инфо», ставящий перед собой цели развития и пропаганды средств программирования с использованием Форт-систем.

Уже сейчас советский пользователь может получить в свое распоряжение обширный набор базовых программных средств для работы на Форте и с помощью Форта. Из них следует особо выделить инструментальные средства, предназначенные для разработки программ на Форте: текстовые редакторы, компиляторы, целевые компиляторы, трансляторы с нескольких языков классического типа (с Паскаля и Бейсика, разрабатывается транслятор для языка СИ).

Подготовлена также первая очередь прикладных Форт-пакетов, включающая средства для работы с графикой, специальный пакет для выполнения действий «вещественной арифметики» — над числами, заданными в формате «плавающей точки», ассемблеры для различных ЭВМ, базу данных, перекодировщики, систему многооконного пользовательского интерфейса и др. Начали уже появляться различные обучающие и демонстрационные системы: учебники по Форту, система «АстроГлаз», предоставляющая пользователю движущуюся карту звездного неба, система «ФортМузыка».

Бурно развиваются промышленные приложения Форт-систем. Кроме указанного ранее использования в качестве инструментального средства для трансляторов с алгоритмических языков, Форт

удобен и для промышленных процессоров, входящих в системы управления и работающих в реальном масштабе времени. Речь идет о процессорах, встроенных в оборудование, в станки с числовым программным управлением, бортовое оборудование летательных аппаратов, устройства и сети связи и т. п. Сюда же относятся управляющие процессоры, координирующие действия систем контролеров в энергетических установках и другом сложном оборудовании. Важной разновидностью таких систем являются установки для тестирования сложной продукции.

Следует упомянуть еще об одном развитии идеи Форта, не успевшем попасть в книгу Л. Броуди. И за рубежом, и в нашей стране успешно конструируются специальные «Форт-процессоры» — вычислительные микросхемы, в которые многие базовые операции языка Форт входят как машинные команды. Высокие технические характеристики уже разработанных процессоров побуждают к еще большему развитию и программных Форт-систем.

И. В. Романовский, д-р физ.-мат. наук, проф.

ПРЕДИСЛОВИЕ

Выход в свет настоящей книги является для сторонников Форта важным событием. В нее вложено столько труда, сил и средств, сколько ни в одно предшествовавшее ей вводное руководство. Мне как автору Форта весьма приятно такое свидетельство роста популярности этого языка программирования.

Я разрабатывал Форт в течение нескольких лет как интерфейс между мной и компьютерами, для которых мне пришлось составлять программы. Традиционные языки программирования не удовлетворяли меня по своим изобразительным средствам, простоте и гибкости, поэтому, чтобы понять, что же нужно программисту для создания эффективных программ, я отказался от многих общепринятых правил. Наиболее существенной мне представляется возможность дополнять язык любыми средствами по мере необходимости.

Свои первые результаты я получил в то время, когда работал с моделью IBM 1130 —ЭВМ «третьего поколения». Они показались мне настолько значительными, что я посчитал новый язык «языком машин четвертого (fourth) поколения» и назвал бы его FOURTH (четвертым), если бы модель 1130 допускала пятисимвольные идентификаторы. Таким образом FOURTH превратился в FORTH[1] (Форт) — своеобразная игра слов.

«Сохранять простоту языка» — так примерно формулируется основной принцип, которым я руководствовался в процессе работы над Фортом и продолжаю руководствоваться при его применении. Элегантность решения — в его простоте. Простота достигается при глубоком изучении *существа* проблемы и осознается при получении правильного решения. Это положение следует особо подчеркнуть, поскольку оно противоречит устоявшемуся мнению о том, что мощность языка увеличивается с возрастанием его сложности. Про-

[1] Вперед. — *Примеч. пер.*

стота гарантирует надежность, компактность и быстродействие Форт-программ, а значит, и доверие к ним пользователей.

Предлагаемая вниманию читателей книга написана и проиллюстрирована Л. Броуди, в высшей степени одаренным человеком с хорошо развитой интуицией и творческим воображением, что становится очевидным в процессе освоения материала. Эта книга представляет собой начальное, но обстоятельное пособие для изучения языка. Автор искусно подводит новичка к тому уровню знаний, который необходим всем программирующим на Форте.

Хотя я — единственный человек, перед которым никогда не стояла задача изучить Форт, мне хорошо известно, с какими это сопряжено трудностями. Как и при изучении любого естественного языка, приходится запоминать правила употребления множества слов. Шутливые пояснения и великолепно подобранные примеры существенно облегчают задачу для начинающих. Для тех же, кто уже имел дело с Фортом, просмотр книги будет подобен приятной прогулке и поможет освежить в памяти «знакомый пейзаж». Однако книга не настолько проста и популярна, чтобы показаться тривиальной. Безусловно, читая ее, вы иногда можете испытывать затруднения, но непременно узнаете много нового и о компьютерах и компиляторах, впрочем, как и о программировании.

Форт предоставляет человеку удобные средства общения с «умными» машинами, которыми он себя окружает. Следовательно, этот язык должен обладать свойствами естественных языков, включая компактность, разносторонность и расширяемость. Я считаю его лучшим языком для написания программ, изложения алгоритмов или объяснения работы ЭВМ и думаю, что, когда вы прочтете эту книгу, вам придется со мной согласиться.

Чарльз Мур, автор языка Форт

ОТ АВТОРА

В 1980 г., когда готовилось первое издание книги, популярность Форта была еще не столь велика. Сегодня положение существенно изменилось. И если интерес к другим языкам то возрастал, то падал (пик популярности Паскаля прошел, на какое-то время воцарился язык Си, но уже главным претендентом «на трон» является Модула 2), к Форту он рос стабильно год от года. Несмотря на то что распространение Форта продолжается в свойственном для него темпе, выпуск Форт-процессора Ч. Мура, как нам кажется, вызовет новый взрыв эмоций у инженеров и программистов.

Моей книге посчастливилось появиться в период расцвета популярности Форта. Первое ее издание переведено на немецкий, французский, японский, датский и китайский языки. За ваше одобрение и поддержку я благодарен вам, мои читатели.

Уже после выхода первого издания был принят и получил признание Стандарт Форт-83. Это явилось главной причиной, побудившей нас приступить к переработке книги, так как она не соответствовала новому стандарту. Что касается синтаксиса, то, поскольку в настоящее время еще существует множество систем, написанных на Фиг-Форте, я включил в текст второго издания сноски для пользователей таких систем, где указаны расхождения со Стандартом-83. В остальном же я старался следовать более строгому стилю написания программ и принципам, изложенным в моей книге «Думаем на Форте» (Brodie, Leo, Thinking Forth. Englewood Cliffs, N. I.: Prentice — Hall, 1984). Необходимо также отметить, что мы перестали употреблять слово *экран* вместо слова «блок». Если ранее такое разделение имело смысл («экран» обычно рассматривается как «блок», содержащий исходный текст), то теперь многие начинающие приходят в замешательство от наличия двух терминов.

В заключение мне хотелось бы выразить благодарность всем, кто так или иначе принимал участие в подготовке первого издания книги: Д. Сэндерсону, М. ЛаМанне, Дж. Дьювею,

Э. К. Конклину, Э. Д. Разер — за консультации по технике и стилю программирования на Форте, К. Хэррису — за советы по методике обучения Форту и ряд написанных им упражнений, К. А. Розенберг — за редактирование и профессиональные замечания, а также за большую работу по форматизации страниц, С. Линстрот, К. Льюбисих, К. Уэвер, К. Кэролин, С. Б. Броуди, Д. Робертсу, Дж. Ритсчер, Д. Разер, В. Шоуз, Н. Эльберт, Б. Робертс, Дж. Дотсону, Б. Патерсону и Г. Фридлендеру — за техническую помощь. Я весьма признателен С. Б. Броуди за ценные критические замечания и конструктивные предложения и, конечно, Ч. Ш. Муру за создание Форта.

При подготовке второго издания существенную помощь мне оказали Дж. Дж. Кэссэди, представивший исходную версию ассемблера 8080, К. Хамбэкер, отрецензировавший переработанную рукопись книги, и М. Хэм, приславший мне множество писем с критическими замечаниями.

Лео Броуди

КОРОТКО О КНИГЕ

Мы начинаем изучать Форт — интересный и мощный машинный язык. Если вы новичок, желающий поближе познакомиться с компьютером, то Форт поможет вам в этом деле. Он в большей степени приспособлен для написания программ, чем любой другой язык (см. «Введение для начинающих»). Если вы умудренный опытом профессионал, который хочет изучить Форт, вам тоже нужна именно наша книга. Форт настолько отличается от остальных языков, что всем, от новичка до специалиста, рекомендуется изучать его с самого начала. Поэтому, если вы знаете другие языки программирования, забудьте их и оставьте в памяти только то, что вам известно о компьютере (см. «Введение для профессионалов»).

Поскольку книга предназначена для читателей с различными уровнями подготовки, она построена таким образом, чтобы можно было знакомиться лишь с тем материалом, который вам необходим. В тексте даются сноски, адресованные разным категориям читателей. Первая половина гл. 7 содержит основы машинной арифметики только для начинающих.

В книге объясняется, как писать простые прикладные программы на Форте. В нее включены все служебные слова языка, требуемые для разработки высокоуровневой прикладной программы в однозадачном режиме, — от команд, реализующих простые математические операции, до команд управления трансляцией. Команды, относящиеся к средствам мультипрограммирования, утилитам вывода на печать и обмена с дисками, а также к объектному компилятору, здесь опущены. Эти команды доступны в некоторых версиях Форта, например в полиФорте. Я подобрал такие примеры программ, которые будут работать в Форт-системе при вводе данных с терминала и диска. Однако не следует считать, что использование Форта ограничивается задачами манипулирования со строками, — сфера его применения гораздо шире.

Как уже отмечалось, книга построена таким образом, чтобы максимально облегчить изучение языка. Все команды описываются дважды: первый раз — в том разделе, где они вводятся, и второй — в конце главы, где дается краткий обзор ее содержания. В приложении Б представлен указатель слов Форта в алфавитном порядке, а в приложении В они сгруппированы по областям применения. В конце каждой главы приводятся, кроме того, словарь терминов и упражнения, ответы на которые вы найдете в приложении А. В процессе изложения даются полезные рекомендации и предлагаются необязательные программы. Последние носят чисто иллюстративный характер и поэтому представлены здесь без каких-либо пояснений.

Следует отметить, что Форт — необычный язык. Он «попирает» многие устоявшиеся правила программирования. Первоначально я воспринял Форт крайне скептически, но по мере создания сложных прикладных программ мне начали открываться его красота и мощь. Постарайтесь относиться к нему без предубеждения, если вам что-то покажется странным. Лишь немногие программисты, освоившие Форт, возвращались снова к другим языкам программирования.

Желаю вам успехов в освоении Форта и его последующем применении.

Лео Броуди

ВВЕДЕНИЕ

ЧТО ТАКОЕ МАШИННЫЙ ЯЗЫК? (ВВЕДЕНИЕ ДЛЯ НАЧИНАЮЩИХ)

Новичок, впервые столкнувшийся с термином «машинный язык», может подумать: «На каком же таком языке разговаривает компьютер? Наверное, человеку чрезвычайно трудно его понять. Выглядит этот язык, вероятно, как-нибудь так:

```
976#!@NX714&+
```

если он вообще как-то выглядит». На самом деле машинный язык не должен быть трудным для понимания. Его назначение — служить удобным средством связи между человеком и компьютером.

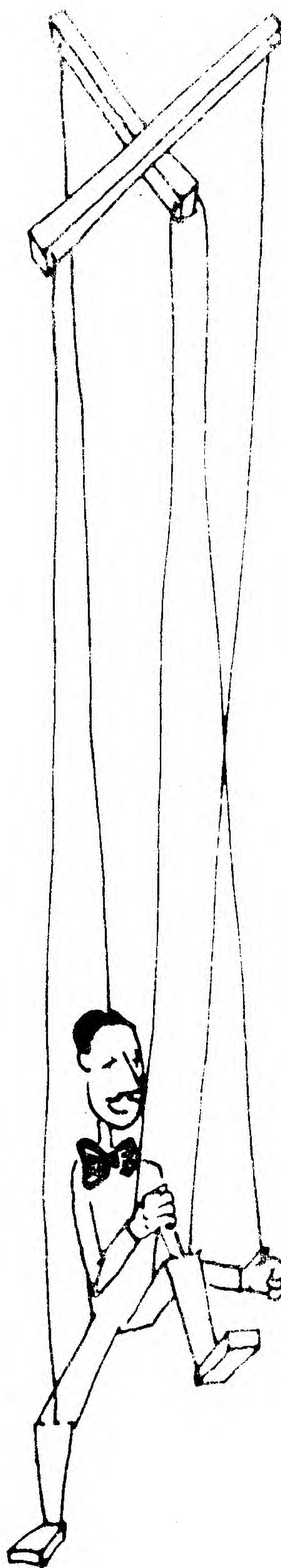
Здесь уместно провести аналогию с марионеткой. Вы можете заставить марионетку «ходить», манипулируя деревянным приспособлением, даже не касаясь нитей, приводящих ее в движение. Эти манипуляции означают «ходьбу» на языке марионетки. Кукольник управляет марионеткой таким способом, который понятен марионетке и легко осуществим кукольником.

Компьютеры — это машины, подобные марионеткам. Ими нужно управлять, пользуясь специальным языком. И поэтому нам необходим язык, обладающий двумя на первый взгляд противоположными свойствами. С одной стороны, он должен точно выражать смысл приказа компьютеру, передавая последнему всю требуемую для выполнения операции информацию, а с другой — быть предельно простым.

Со временем появления компьютеров было разработано множество языков: Фортран, который считается старейшиной среди них, Кобол — образец языка для обработки коммерческой информации, Бейсик, предназначенный для тех, кто делает первые шаги на пути изучения таких языков, как Фортран и Кобол. Предметом нашей книги является язык, совершенно не похожий на другие: Форт. Популярность Форта непрерывно возрастает в течение

последних нескольких лет; причем во всех областях программирования.

Упомянутые выше языки, включая Форт, относятся к языкам высокого уровня. Начинающему важно понять разницу между языком высокого уровня и языком, понятным компьютеру. Все языки высокого уровня выглядят для программиста одинаково, независимо от того, на компьютере какой марки или модели будет выполняться соответствующая программа. Но каждый компьютер имеет свой внутренний, или «машинный», язык. Что же представляет собой машинный язык? Обратимся снова к примеру с марионеткой.



Вообразите, что деревянное приспособление для управления марионеткой отсутствует, и кукольник непосредственно держит в руках нити, каждая из которых связана с одной частью тела марионетки. Согласованная комбинация движения отдельных нитей может считаться «машинным языком» нашей марионетки. Теперь представьте себе, что нити привязаны к деревянному приспособлению. Это приспособление соответствует языку высокого уровня.

Простым поворотом запястья кукольник может управлять одновременно множеством нитей.

Точно так же в языке высокого уровня знакомый вам знак «+» инициирует выполнение множества внутренних функций, в результате чего производится суммирование. Компьютер может быть запрограммирован на перевод символов языка высокого уровня (таких, как «+») в свой собственный машинный язык, после чего он может выполнить полученные машинные команды.

Итак, язык высокого уровня — это средство записи программы символами и словами, понимаемыми человеком, которые затем переводятся на машинный язык компьютера конкретной марки или модели.



В чем же состоит отличие Форта от других языков высокого уровня? Это отличие в том, как он разрешает противоречие между человеком и машиной. Язык должен быть удобным для человека, но в то же время соответствовать операциям, выполняемым компьютером. Форт — единственный в своем роде язык, где данная проблема решена уникально. Каким образом она решена, будет показано в дальнейшем.

ОБЛАСТИ ПРИМЕНЕНИЯ ФОРТА (ВВЕДЕНИЕ ДЛЯ ПРОФЕССИОНАЛОВ)

Особенно популярен Форт стал с 1978 г., хотя применялся в основных областях науки, экономики и производства с начала 70-х годов. Где бы вы ни работали, скорее всего, ваша прикладная программа, написанная на Форте, будет выполняться более эффективно, чем на том языке, который вы применяете сейчас. Для того чтобы это понять, вы должны прочитать настоящую книгу, по возможности найти Форт-систему и попытаться с ней поэкспериментировать. В данном разделе вы найдете для себя ответы на два вопроса: «Что такое Форт?» и «Где он может использоваться?».

Форт многогранен. Его можно рассматривать как

- язык высокого уровня;
- язык Ассемблера;
- операционную систему;
- инструментарий для создания программ;
- некоторую концепцию разработки программного обеспечения.

Форт как язык начинается с мощного набора стандартных команд, образующих механизм, с помощью которого вы можете формировать свои собственные команды. Процесс построения определений по модульному принципу — черта, объединяющая Форт с языками высокого уровня. С другой стороны, команды Форта могут быть определены непосредственно на уровне мнемоники ассемблера с помощью ассемблера Форта. Все команды интерпретируются одним и тем же интерпретатором и компилируются одним и тем же компилятором, что придает языку поразительную гибкость. Кодирование вашей программы на самом высоком уровне будет подобно записи этой программы на некотором подмножестве естественного языка. Форт получил название «метаприкладного» языка, так как он позволяет создавать проблемно-ориентированные языки.

Вы можете разбивать задачу на небольшие фрагменты, создавать слова с небольшими определениями, реализующие эти фрагменты, а затем объединять созданные слова небольшими порциями в другие слова. Именно такой подход присущ человеку

в его деятельности. На каждом этапе своей работы программист оперирует лишь несколькими понятиями, соответствующими возможностям кратковременной памяти человека. При использовании этих понятий на последующих этапах мы обращаемся к ним по имени, что соответствует образу мышления человека.

Диалог — неотъемлемое свойство Форта. Новые слова могут быть скомпилированы таким образом, что программист получит возможность сразу же проверять каждую новую команду и следить за состоянием своей программы посредством немедленной обратной связи. (Во многих языках программирования для этого требуется загрузка текстового редактора, редактирование, выход из текстового редактора, загрузка компилятора, компиляция, выход из компилятора, загрузка редактора связей и т. д.) *Итерационный подход*, при котором оптимальное решение находится в процессе тестирования программных моделей, наиболее приспособлен к интегрированной среде с небольшим временем реагирования на запросы, а именно такую среду и обеспечивает Форт.

Структурные операторы управления Форта вынуждают создавать программу с вложенными структурами, что уменьшает сложность программы. А поскольку реализация Форта ориентирована на вызов слов, то пользователю предпочтительнее работать с небольшими подпрограммами (словами), причем даже с меньшими, чем это позволяют делать традиционные «модульные» языки программирования. Практически без потери эффективности Форт способствует «упрятыванию» информации, что в свою очередь упрощает модернизацию программы. Вследствие этого по имеющимся данным сокращается время разработки Форт-программ: по сравнению с Ассемблер-программами на порядок, а по сравнению с программами на языках высокого уровня — в два раза.

Форт не только увеличивает производительность программирования, но и повышает скорость работы ваших программ. Форт гарантирует быстродействие. Программа, написанная на Форте высокого уровня, выполняется быстрее, чем большинство программ, написанных на других языках высокого уровня; ее производительность составляет почти половину производительности программ, написанных на языке Ассемблера. Критичные по времени фрагменты программ вы можете писать на ассемблере Форта, и они будут выполняться со скоростью, обеспечиваемой процессором.

Помимо всего код Форта компактен. Прикладные программы, написанные на Форте, занимают меньший объем памяти, чем аналогичные программы, созданные с помощью традиционного *Ассемблера*! Написанная на Форте операционная система вместе со стандартным набором слов занимает менее 8K байт. Вся среда Форта спокойно умещается в пространстве, составляющем 16-32K.

Динамическая среда для объектной прикладной программы может потребовать объем памяти менее 1К байт.

Форт мобилен. Виртуальная Форт-машина реализована почти на всех известных мини- и микрокомпьютерах. На самом деле Форт-архитектура уже воплощена в кремнии.

Ниже приводится несколько примеров применения Форта.

Искусство. Форт используется для управления оборудованием при съемках видеоклипов некоторых рок-групп, а также бегущей строкой рекламы шоу Била Косби. В музыкальной студии электронных инструментов Государственного университета Сан-Хосе язык MASC, являющийся расширением Форта, позволяет композиторам писать музыку для аналоговых синтезаторов.

Программное обеспечение для персональных компьютеров и бизнеса. Форт применялся при разработке учетных программ «Назад к Бейсику» фирмы Peachtree, пакета СУБД фирмы Savvy, системы Симплекс фирмы Quest Research (интегрированная база данных со средствами текстовой обработки и передачи сообщений, макинтошеподобными окнами и графикой) и комплексов Дэйта Эйс и Мастер Тайп. Фирма Bell Canada выбрала Форт для реализации программ аварийной диагностики телефонной связи и сети баз данных, где единственный процессор 68000 обслуживает 32 терминала и базу данных объемом 200 МБ. Фирма Cycledata использует Форт для создания и сопровождения базы данных курса акций с выдачей информации клиентам в реальном времени.

Сбор и анализ данных. Форт применяется во многих крупных обсерваториях планеты. Например, компьютер PDP-11/34 с Форт-системой полностью управляет обсерваторией, в том числе телескопом, куполом, несколькими электронно-лучевыми трубками, строчнопечатающим устройством, дисководами с гибкими дисками, и в то же время обеспечивает сбор данных по инфракрасному излучению из космоса, анализ этих данных и выдачу результатов на графический монитор. Лесная служба США с помощью системы распознавания образов, написанной на Форте, производит анализ и переработку контурных карт. Фирма Dysan применяет написанную на Форте инструментальную систему управления качеством, которая в среде IBM PC работает с битовыми шаблонами, хранящимися на гибких дисках. Устройства измерения глубины, управляемые Фортом, используются на буксирах Миссисипи. NASA, Центр систем океана ВМС США и Центр вооружений ВМС США применяют Форт для различных видов сложного анализа данных. В прикладные программы такого рода часто включаются написанные на Форте программы быстрого преобразования Фурье и Уолша, численного интегрирования, а также Форт-программы, реализующие другие математические методы.

Экспертные системы. Для компании General Electric Corporate Research and Development на Форте была создана экспертная си-

стема диагностики дизель-электровозов. Форт также использовался компанией Applied Intelligence Systems and IRI, Inc. для промышленного прогнозирования. Центр исследований проблем сна Станфордского университета применяет созданную на Форте экспертную систему в целях идентификации моделей сна.

Графика. Программа Изель и ее приемник Люмена, созданные фирмой Time Arts, Inc., — программы-художники. Обе написаны на Форте и продаются в сочетании с известными графическими системами и системами автоматизированного проектирования.

Медицина. Единственный в Главном госпитале компьютер PDP-11 дает возможность одновременно обслуживать большую базу данных, где хранится информация о пациентах, управлять 32 терминалами и оптическим считывателем, делать анализ крови и измерять пульс больного в реальном времени, осуществлять статистический анализ информации из базы данных для установления соответствия между физическими симптомами заболевания, правильностью диагноза и результатами лечения. Отдел медицинских систем NCR в своей системе 9300, применяемой в больничном обслуживании, также используют Форт. Административный центр по исследованию и совершенствованию реабилитации ветеранов применяет Форт для создания устройств обслуживания инвалидов, включая ультразвуковой детектор, который переводит команды, посылаемые человеком (кивок, поворот головы), в сигналы управления устройством, например креслом на колесах с электроприводом.

Переносные «разумные» устройства. Существует множество разных приборов с встроенными Форт-программами: прибор для диагностики заболеваний сердечно-сосудистой системы, самоходный датчик воспламенения, созданный двумя компаниями, прибор для определения относительной влажности различных сортов зерна, транслятор с языка Craig и т. д.

Управление процессами. Лаборатория Jet Propulsion и компания McDonnel Douglas Astronoutics независимо друг от друга выбрали Форт для разработки приборов, применяющихся при создании промышленных материалов в условиях невесомости. Lockheed California и TRW каждая по-своему используют Форт при создании радарных антенн. Фирма Northrup применяет Форт в качестве стандартного языка тестирования. С помощью Форта фирма Johnson Filaments управляет лазерным микроизмерительным роботом при измерении диаметра пластиковых волокон. На предприятиях Union Carbide Форт используется при разработке лабораторных средств автоматизации и создании автоматизированных систем управления процессами.

Роботы. Управляемая голосом сервосистема, применяемая на TRW, написана на Форте. С помощью подвесной автоматической видеокамеры Скайкэм осуществляется трансляция футбольных

матчей, а камера фирмы Elicon создает фрагменты для повторного показа. Диапазон других применений Форта — от управления разгрузкой и погрузкой багажа на главной авиалинии США до сортировки персиков на Калифорнийском консервном заводе.

Форт-процессоры. R65F11 и R65F12, созданные Rockwell International Corp., представляют собой восьмиразрядные процессоры, размещающие 133 Форт-слова на одном кристалле ПЗУ. Вспомогательные кристаллы ПЗУ содержат дополнительные слова Форта, полезные для создания программного обеспечения. Семейство MA2000 National Semiconductor Corp. состоит из набора модулей с интерфейсом через стек, формирующих в совокупности полный автономный Форт-компьютер. Семейство высокоскоростных процессоров Novix NC4000 представляет собой Форт-кристаллы, в которых команды Форта высокого уровня выполняются за один такт.

Завершая введение, хотелось бы обратить ваше внимание на одну особенность Форта. Дело в том, что ответственность за производительность центрального процессора (ЦП) возлагается *на вас*. Можно провести следующую аналогию. Водителю автомашины ручное управление труднее освоить, чем автоматическое, но все-таки ручное управление позволяет вести автомашину лучше. Точно так же специалисту труднее изучить Форт, чем традиционные языки высокого уровня, похожие друг на друга (освоив один из них, вы легко можете выучить другой). Но уж если вы однажды выучили Форт, то это даст вам возможность экономно расходовать машинное время и память, а также внедрить новую технологию, с помощью которой вы сможете значительно сократить сроки разработки проекта. И помните, что все компоненты Форта, включая операционную систему, компилятор, интерпретаторы, текстовый редактор, виртуальную память, ассемблер и средства мультипрограммирования, следуют одному и тому же протоколу. Путь к Форту короче, чем изучение по отдельности перечисленных выше компонент.

Если все изложенное здесь вас заинтересовало, значит, вы уже готовы приступить к изучению Форта.

Глава 1.
ОСНОВЫ ФОРТА

В настоящей главе вы познакомитесь с некоторыми особенностями Форта. Прочитав несколько вводных страниц, вы уже сможете сесть за терминал. Если же у вас нет терминала, не огорчайтесь — в процессе изложения мы шаг за шагом будем выдавать вам результат.

ЖИВОЙ ЯЗЫК

Вообразите, что вы — управляющий офисом и только что взяли себе нового энергичного помощника. В первый день вы учите его оформлять вашу корреспонденцию по соответствующему формату (допустим, ваш помощник уже умеет печатать на машинке). В конце дня вы уже можете просто сказать ему: «Пожалуйста, отпечатайте это». На второй день вы показываете своему помощнику систему регистрации. Все утро у вас уходит на соответствующие разъяснения, но уже после обеда вам достаточно отдать короткое распоряжение: «Пожалуйста, зарегистрируйте это». К концу недели вы будете понимать друг друга с полуслова. Так, если вы скажете своему помощнику: «Пожалуйста, отправьте это письмо», он поймет, что письмо нужно напечатать, дать вам на подпись, сделать с него фотокопию и зарегистрировать ее, а оригинал отослать по почте. Подобные взаимоотношения позволят как вам, так и вашему помощнику выполнять свою работу более эффективно и доставят вам обоим больше удовольствия, чем любые другие.

Итак, для правильной организации труда и обеспечения эффективного взаимодействия сотрудников необходимо:

- определить круг задач и присвоить каждой задаче имя;
- сгруппировать однотипные задачи в более крупные и присвоить каждой из укрупненных задач имя и т. д.

Форт дает вам возможность аналогичным образом организовать ваши собственные процедуры и передать их компьютеру таким же способом (разве что не говорить ему: «пожалуйста»). В качестве примера можно привести стиральную машину, управляемую микропроцессором с программой на Форте. Заключительной командой в нашем примере будет команда, которой мы присвоим имя СТИРАЛЬНАЯ-МАШИНА. Ниже дается определение команды СТИРАЛЬНАЯ-МАШИНА так, как оно выглядит на Форте:

```
: СТИРАЛЬНАЯ-МАШИНА  СТИРАТЬ ВЫКРУЧИВАТЬ ПОЛОСКАТЬ ВЫКРУЧИВАТЬ ;
```

На языке Форт двоеточие означает начало нового определения. Первое слово после двоеточия, СТИРАЛЬНАЯ-МАШИНА, является именем новой процедуры. Остальные слова, СТИРАТЬ, ВЫКРУЧИВАТЬ, ПОЛОСКАТЬ, ВЫКРУЧИВАТЬ, составляют «определение» этой новой процедуры. Наконец, точкой с запятой отмечается конец определения.



Каждое слово, входящее в состав определения СТИРАЛЬНАЯ-МАШИНА в нашей программе, описывающей стиральную машину, уже специфицировано. В частности, посмотрим, как записывается определение команды ПОЛОСКАТЬ.

```
: ПОЛОСКАТЬ  НАЛИТЬ-ВОДУ СТИРАТЬ ВЫЛИТЬ-ВОДУ ;
```

Как видите, определение ПОЛОСКАТЬ состоит из группы слов: НАЛИТЬ-ВОДУ, СТИРАТЬ и ВЫЛИТЬ-ВОДУ. Опять-таки каждое из этих слов уже где-то специфицировано в программе, описывающей стиральную машину. Определение команды НАЛИТЬ-ВОДУ может быть таким:

```
: НАЛИТЬ-ВОДУ  КРАНЫ ОТКРЫТЬ ДО-НАПОЛНЕНИЯ КРАНЫ ЗАКРЫТЬ ;
```

В приведенном определении мы ссылаемся как на *объекты* (краны), так и на *действия* (открыть и закрыть). Слово ДО-НАПОЛНЕНИЯ введено для создания «Цикла задержки», чтобы контролировать включение индикатора уровня заполнения емкости стиральной машины водой.

Если мы проследим эти определения в обратном порядке, то в конечном итоге обнаружим, что все они специфицированы в терминах группы команд, которые образуют основу всех Форт-

систем. Например, полиФорт включает около 300 таких команд. Одни из них сами определены через двоеточие, как было показано выше, другие — непосредственно в терминах машинного языка конкретного компьютера. В языке Форт специфицированная таким образом команда называется *словом*.

Возможность определять слова в терминах других слов называется *расширяемостью*. Расширяемость является основой хорошего стиля программирования и позволяет достичь необходимого уровня мощности языка.

Обслуживает ли ваша программа линию монтажа, собирает ли данные для научного эксперимента, используется ли для коммерческих целей или является игровой — во всех случаях вы можете создать свой собственный «живой язык», который соответствует вашим потребностям.

В этой книге мы рассмотрим большинство часто используемых стандартных команд Форта.

ДИАЛОГ

Одно из специфических свойств Форта состоит в том, что он дает возможность «выполнить» слово, просто написав его. Достаточно просто набрать это слово на клавиатуре и нажать клавишу RETURN (возврат каретки или ввод). Конечно, можно применять данное слово в определении других слов, помещая его в соответствующее определение.

Форт называется *диалоговым* языком, потому что его команды выполняются сразу, как только вы их вводите. В качестве примера (вы можете его выполнить самостоятельно) рассмотрим процесс объединения простых команд в более сложные. Мы будем использовать некоторые слова Форта для управления экраном дисплея или печатающим устройством. Но прежде познакомимся с механизмом «диалога» посредством клавиатуры терминала.

Займите место за своим терминалом (для некоторых он будет воображаемым). Возможно, кто-нибудь великодушно предоставит вам условия для занятий, в противном случае вам придется внимательно интерпретировать самому все команды, предназначенные вашему компьютеру. Нажмите клавишу RETURN. Компьютер ответит: ok («все в порядке»), что означает приглашение к работе. С помощью клавиши RETURN вы передаете компьютеру свой запрос. Ответ ok свидетельствует о том, что ваш запрос выполнен, причем без единой ошибки. Мы пока ни о чем не просили, поэтому компьютер послушно ничего не выполнил и выдал приглашение ok.

Теперь введите

```
15 SPACES
```

что означает 15 ПРОБЕЛОВ.

Для многих Форт-систем имеет значение, на каком регистре — верхнем или нижнем — вы набираете текст, поэтому, вводя SPACES, убедитесь в том, что набираете эту строку на верхнем регистре. Если во время набора была допущена ошибка, вы можете исправить ее, нажав клавишу backspace (возврат на одну позицию). Вернитесь к тому месту, где сделана ошибка, введите нужный символ и продолжайте набор. Набрав строку правильно, нажмите клавишу RETURN. (После нажатия клавиши RETURN исправлять ошибку уже поздно.)

В дальнейшем мы будем использовать обозначение <return> в тех местах, где вы должны нажимать клавишу RETURN, и подчеркивать ответы компьютера, чтобы отличать их от других символов (даже если сам компьютер и не подчеркивает свои ответы).

Что произойдет в такой ситуации:

```
15 SPACES<return>                ok
```

Как только вы нажали клавишу ввода, компьютер выведет 15 пробелов и затем, выполнив ваш запрос, выдает: ok (после 15-го пробела).

Наберите на клавиатуре следующее:

```
42 EMIT<return> *ok
```

Фраза «42 EMIT» приказывает компьютеру вывести символ * (мы обсудим эту команду позднее). Компьютер выводит требуемый символ, а затем ok

На одной строке мы можем помещать несколько команд. Например:

```
15 SPACES  42 EMIT  42 EMIT<return>                **ok
```

На этот раз компьютер выводит 15 пробелов и две звездочки. Отметим, что при вводе слов и/или чисел их можно разделять любым количеством пробелов (как вам удобно), но между ними должен быть *хотя бы один пробел*, чтобы компьютер мог различать слова и/или числа.

Вместо того чтобы всякий раз вводить фразу

```
42 EMIT
```

давайте определим ее как слово STAR (ЗВЕЗДОЧКА). Итак, введите:

```
: STAR   42 EMIT ;<return> ok
```

Здесь STAR — имя, а 42 EMIT — определение. Заметьте, что мы отделили двоеточие и точку с запятой от соседних с ними слов одним пробелом. Чтобы определения Форта легче воспринимались, условимся отделять имя определения от собственно определения тремя пробелами.

После того как вы наберете на клавиатуре приведенное выше определение и нажмете клавишу RETURN, компьютер ответит вам: ok, т. е. он распознал ваше определение и запомнил его. Введите далее:

```
STAR<return> *ok
```

Как видите, компьютер выполнил ваш приказ и выдал звездочку. Определенное вами слово STAR ничем не отличается от определенного ранее **EMIT**. Поэтому чтобы вам легче было ориентироваться, ранее определенные слова мы будем выделять полужирным шрифтом.

Другим определенным системой словом является **CR**, которое обеспечивает возврат каретки и перевод строки на вашем терминале. Обязательно почувствуйте разницу в использовании клавиши RETURN и словом Форта **CR**. В качестве примера наберите на клавиатуре

```
CR<return>_
ok
```

Компьютер осуществил возврат каретки, а затем вывел ok (на следующей строке).

Введите такой текст:

```
CR STAR CR STAR CR STAR<return>
*
*
*ok
```

Поместим **CR** в определение слова MARGIN (ПОЛЕ ПРОБЕЛОВ):

```
: MARGIN   CR  30 SPACES ;<return> ok
```

Теперь вы можете ввести следующее:

```
MARGIN STAR MARGIN STAR MARGIN STAR<return>
```

и получите три вертикально расположенные звездочки, дополненные слева 30 пробелами.

Комбинация слов MARGIN STAR пригодится нам в дальнейшем, поэтому введем определение BLIP (ТОЧКА):

```
: BLIP   MARGIN STAR ;<return> ok
```

Нам также предстоит выводить горизонтальные последовательности звездочек. Для этой цели введем следующее определение (его назначение мы объясним позднее):

```
: STARS   0 DO STAR LOOP ;<return> ok
```

Итак, мы можем ввести:

```
5 STARS<return> *****ok
```

или

```
35 STARS<return> ***********************************ok
```

или любое представимое число звездочек! Однако нельзя задавать сочетание "0 STARS", в особенности для систем Форта-83. Подробнее об этом речь пойдет в гл. 6.

Нам необходимо слово, которое выполняет команду MARGIN, а затем выводит пять звездочек. Определим слово BAR (ПОЛОСКА):

```
: BAR   MARGIN  5 STARS ;<return> ok
```

после чего можно ввести строку:

```
BAR BLIP BAR BLIP BLIP  CR
```

В результате вы получите букву F, составленную из звездочек:

```
*****
*
*****
*
*
```

В заключение определим слово для этой новой процедуры. Назовем его F:

```
: F   BAR BLIP BAR BLIP BLIP CR ;<return> ok
```

В этом примере показано, каким образом простые команды Форта могут становиться основой для образования более сложных

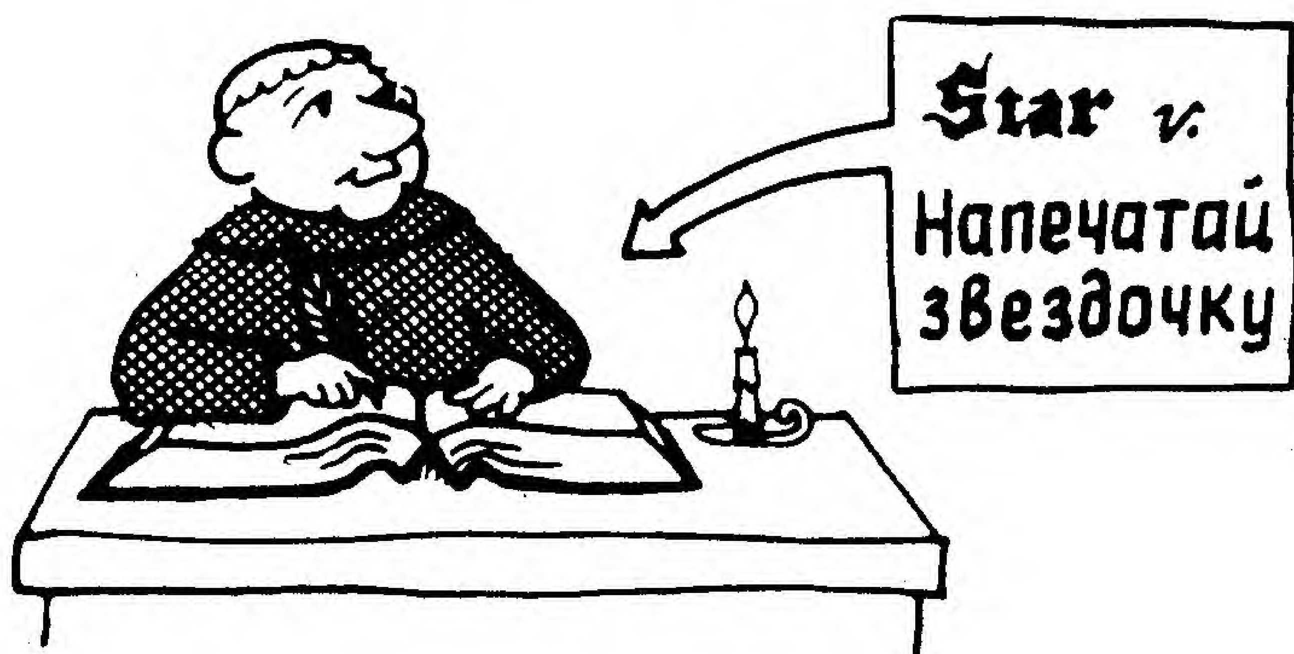
команд. Программа на Форте выглядит скорее как ряд нарастающих по мощности определений, чем как последовательность команд, задающая порядок их выполнения. Чтобы вы имели представление о реальной Форт-программе, мы приводим здесь распечатку нашей учебной программы:

```
( Большая буква F )
: STAR   42 EMIT ;
: STARS   0 DO STAR LOOP ;
: MARGIN   CR 30 SPACES ;
: BLIP   MARGIN STAR ;
: BAR   MARGIN 5 STARS ;
: F   BAR BLIP BAR BLIP BLIP  CR ;
```

СЛОВАРЬ

Каждое слово (его имя и определение) заносится в так называемый словарь Форта. Этот словарь, когда вы начинаете писать на Форте, уже содержит какое-то количество слов, но вы можете помещать в него и «свои» слова. При определении нового слова оно переводится в словарную форму и добавляется в словарь. Такой процесс называется *компиляцией*.

: STAR   42 EMIT ;



К примеру, если вы вводите строку

```
: STAR   42 EMIT ;<return>
```

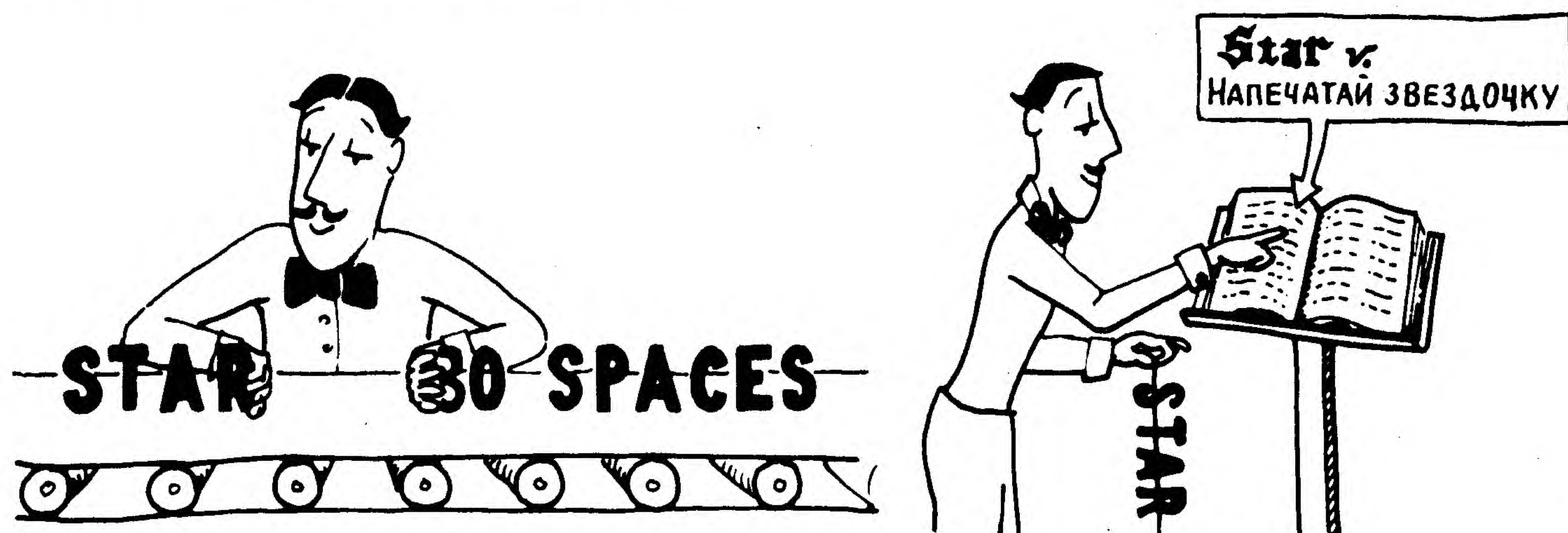
компилятор компилирует новое определение в словарь. Сам компилятор *не* выдает на печать звездочку.

(Не хотите ли вы уже сейчас посмотреть список слов, находящихся в словаре? В большинстве систем для этого нужно ввести слово **WORDS**, и тогда вы получите на экране имена слов вместе с их «адресами» (ссылками на участки памяти). Слова перечисляются в том порядке, в котором их определили — последнее определенное слово будет верхним. В некоторых, более старых системах для таких целей существует слово **VLIST**.)

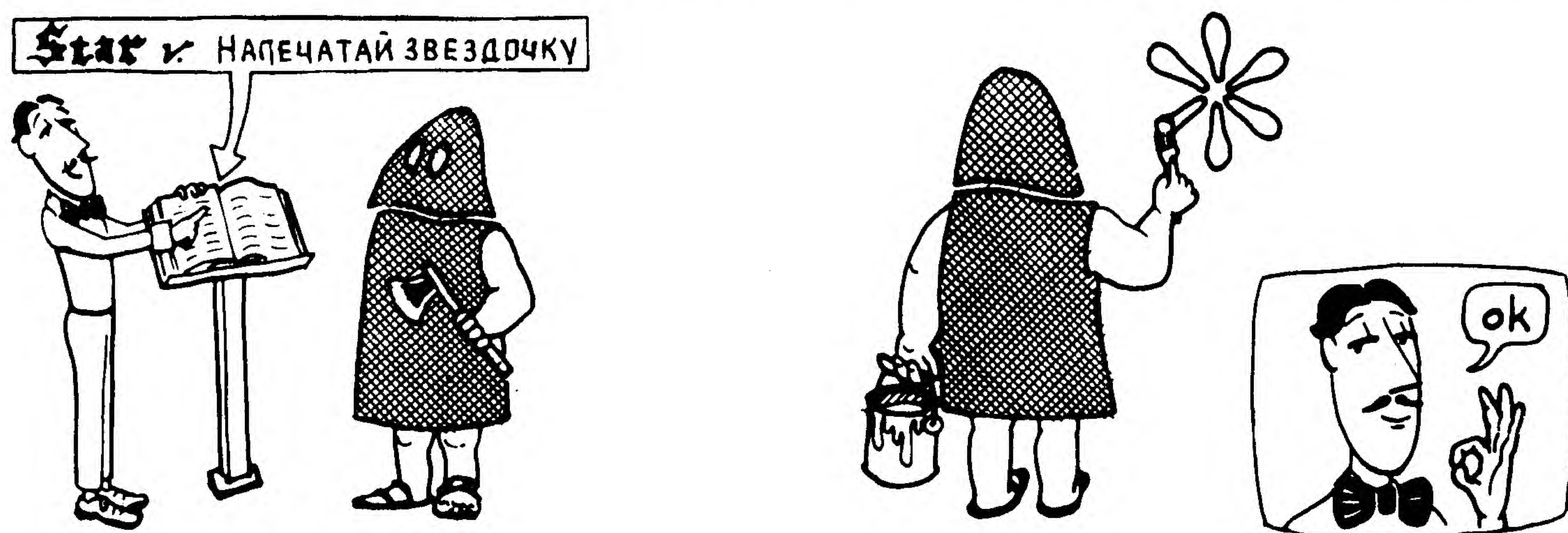
Теперь слово STAR находится в словаре. А как выполняется слово, находящееся в словаре? Наберите непосредственно (не внутри определения) следующую строку:

```
STAR 30 SPACES<return>
```

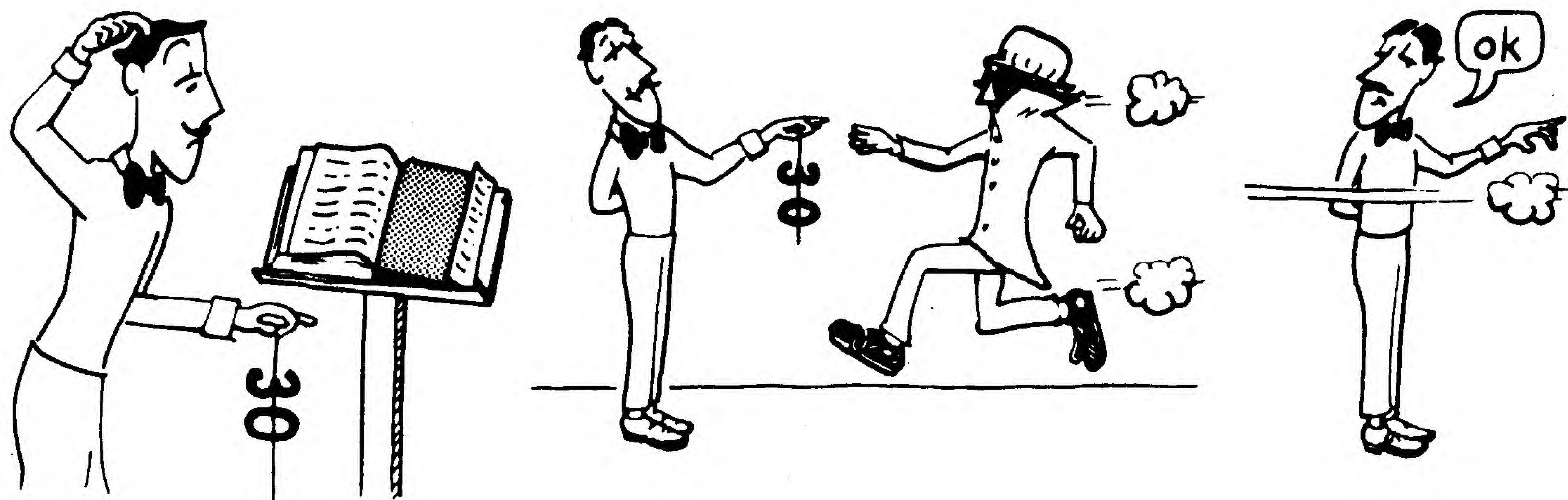
Эта строка активирует слово с именем **INTERPRET**, означающее «текстовый интерпретатор».



Текстовый интерпретатор просматривает входной поток символов в поисках строки символов, внутри которой нет пробелов. Если он находит такую строку, то он ищет ее в словаре.



Найдя нужное слово в словаре, интерпретатор передает его определение слову с именем **EXECUTE**[1], которое выполняет это определение (в нашем случае оно инициирует печать звездочки). Интерпретатор вновь выдает вам ok.



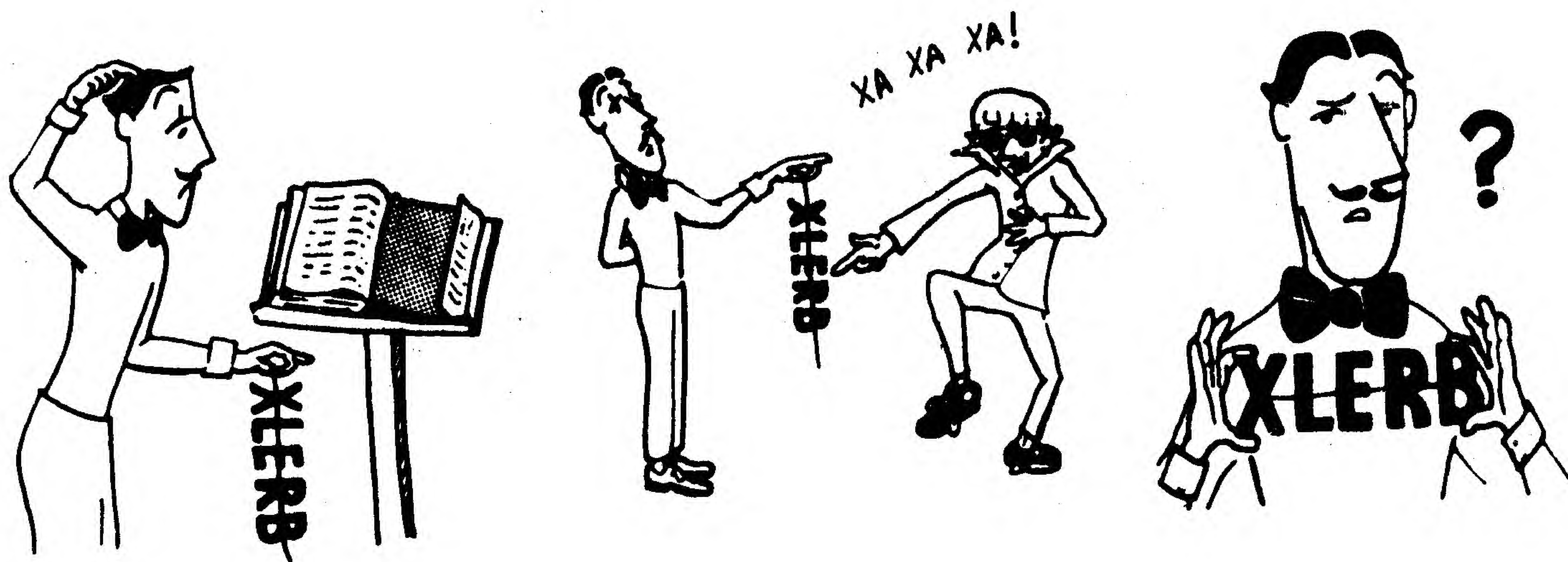
[1] Слово EXECUTE (ВЫПОЛНИТЬ) в одном из вариантов переводится как «казнить». Отсюда ассоциации с палачом. — *Примеч. пер.*

Если интерпретатор не находит такой строки в словаре, он обращается к обработчику чисел (**NUMBER**). Последний проверяет, не является ли переданная ему информация числом, и если это действительно так, помещает его в участок памяти, отведенный для чисел.

Что произойдет, если вы попытаетесь выполнить слово, которого нет в словаре? Введите следующую фразу:

```
XLERB<return> XLERB ?
```

Не найдя слово XLERB в словаре, текстовый интерпретатор попытается передать его **NUMBER**, который не признает его, после чего интерпретатор вернет вам это слово со знаком вопроса.

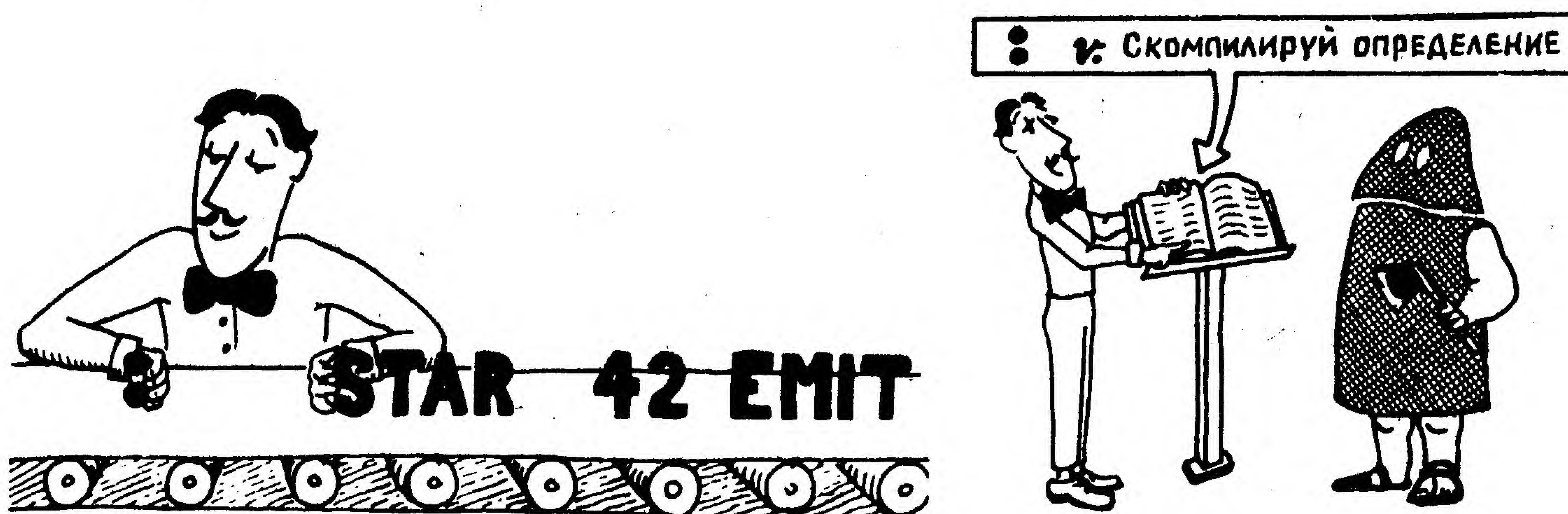


Итак, когда вы набираете на терминале предварительно определенное слово, оно интерпретируется и затем выполняется.

Упоминали ли мы о том, что символ «:» является словом? Так, если вы вводите этот символ:

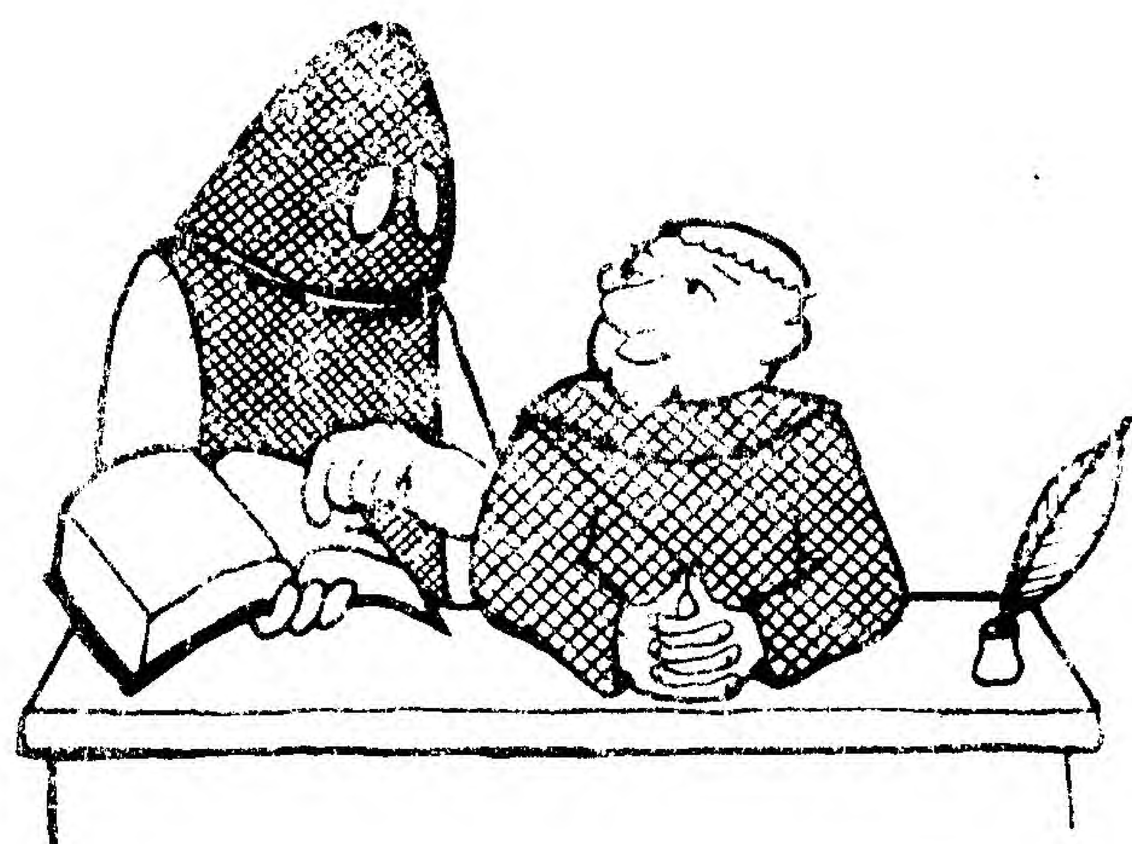
```
: STAR   42 EMIT ;<return>
```

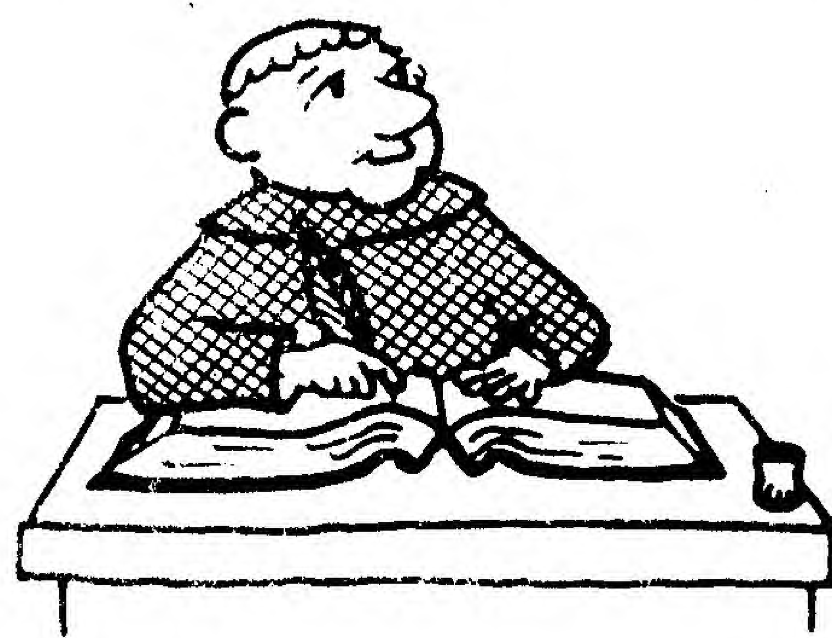
происходит следующее:



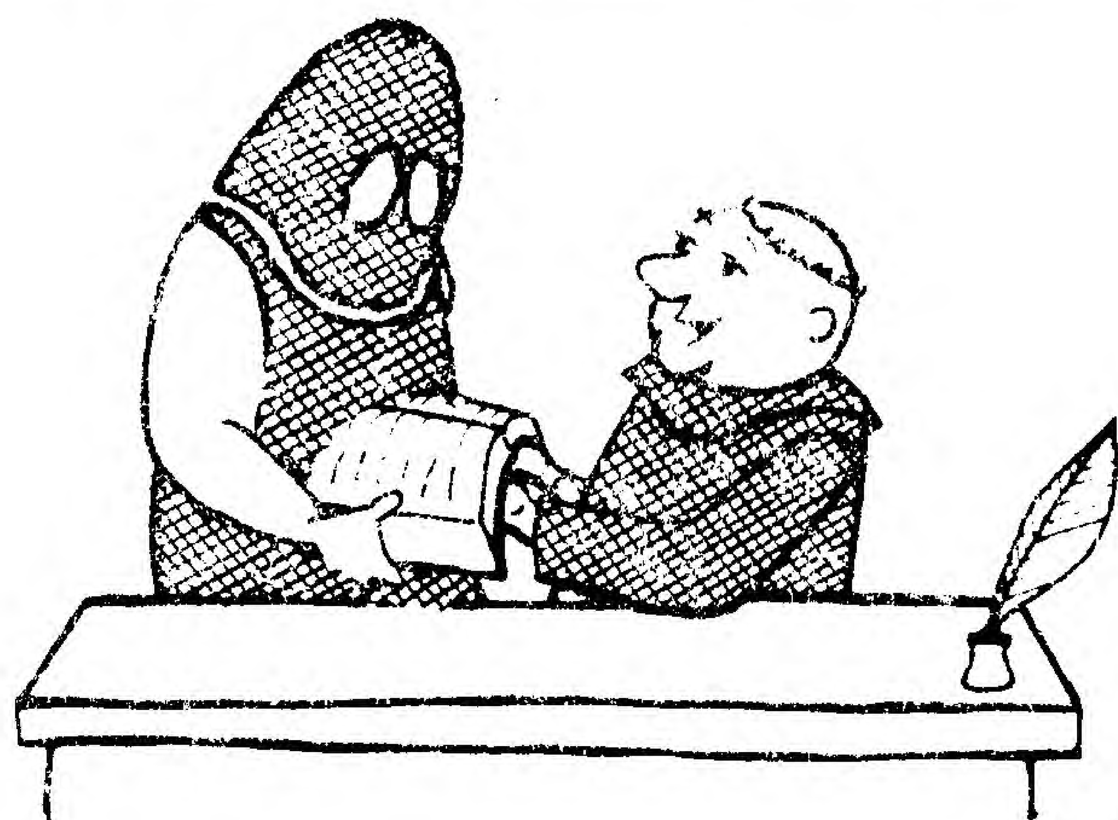
текстовый интерпретатор находит двоеточие во входном потоке и передает его **EXECUTE**.

: STAR 42 EMIT ;





EXECUTE предлагает начать компиляцию. Компилятор переводит определение в словарную форму и записывает его в словарь.





Дойдя до точки с запятой, компилятор останавливается и снова начинает выполняться текстовый интерпретатор, который выдает на экран ok.

КАК ПРАВИЛЬНО «ОБЪЯСНЯТЬСЯ» НА ФОРТЕ!

В Форте словом считается отдельный символ или группа символов, имеющие определение. Почти любое сочетание символов может употребляться для именования слов. Ниже приводятся группы символов, которые нельзя использовать для этих целей, поскольку компьютер может «подумать»[1], что вы собираетесь выполнить одну из следующих операций:

- RETURN — завершить набор символов;
- BACKSPACE — исправить ошибочно набранный символ;
- SPACE (пробел) — указать конец слова.

Теперь разберем слово, имя которого состоит из двух знаков пунктуации. Это слово ." и произносится оно как «точка-кавычка». Вы можете применять его внутри определения для вывода «строки» текста на своем терминале. Например[2]:

---

[1] *Для философов*, конечно, компьютер не «думает», но, к сожалению, очень трудно подобрать слово, точно отражающее его действия. Поэтому условимся считать, что компьютер «думает».

[2] В Форте определен базовый набор слов. В книге такие слова даются на языке оригинала. В большинстве приводимых здесь примеров слова, определяемые пользователем, написаны по-русски. Версии Форта, распространенные в СССР, позволяют это делать. — *Примеч. ред.*

```
: ВСТРЕЧА   ." Привет, я говорю на Форте " ;<return> ok
```

Мы только что определили слово с именем ВСТРЕЧА. Его определение состоит только из одного слова, .", за которым следует текст, подлежащий выводу. Знак кавычки в *конце* текста не будет выведен; он отмечает конец текста и называется *ограничителем*. Специфицируя слово ВСТРЕЧА, не забудьте в конце поставить символ ; — знак конца определения.

Давайте выполним слово ВСТРЕЧА:

```
ВСТРЕЧА<return> Привет, я говорю на Форте ok
```

ПЕРИОД ИСПОЛНЕНИЯ И ПЕРИОД КОМПИЛЯЦИИ

При изучении Форта вам встретятся два термина: *период исполнения и период компиляции*. Смысл их достаточно прост. Если вы компилируете определение некоторого слова, то речь идет о периоде его компиляции, а если вы выполняете слово — о периоде исполнения.

Например, в только что приведенном определении ВСТРЕЧА компьютер во время компиляции выдал вам просто «ok», а во время выполнения указанного определения вывел на экран следующее: «Привет, я говорю на Форте». В случае предварительно определенных слов, таких, как **CR**, период компиляции приходится на время создания Форт-системы, а период выполнения наступает всякий раз, когда вы обращаетесь к этим словам (или к словам, которые в свою очередь обращаются к ним).

Посмотрим на эту ситуацию с другой стороны. Когда мы компилируем слово ВСТРЕЧА, компилятор Форта (или набор так называемых «компилирующих слов», осуществляющих компиляцию) на самом деле исполняется. Поэтому для слова ВСТРЕЧА это период компиляции, а для компилирующих слов Форта — период исполнения.

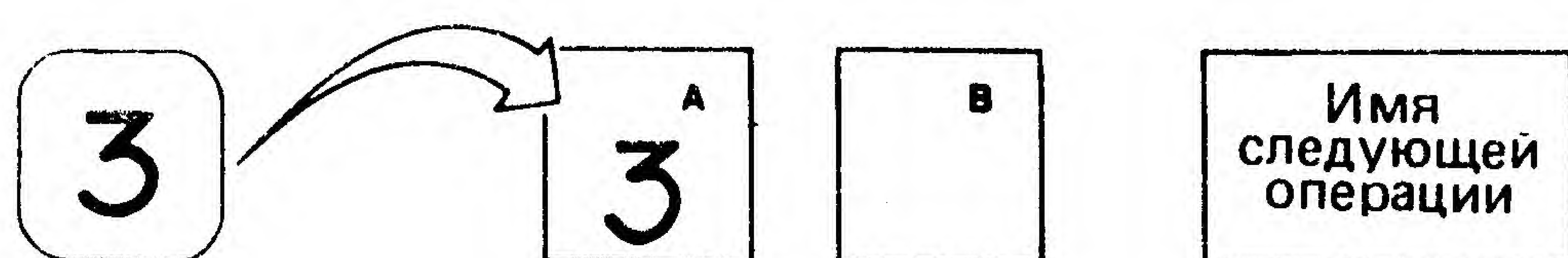
Термины *период компиляции* и *период исполнения* будут вам особенно полезны при рассмотрении более сложных слов.

СТЕК — РАБОЧАЯ ОБЛАСТЬ ОПЕРАТИВНОЙ ПАМЯТИ ДЛЯ ВЫПОЛНЕНИЯ АРИФМЕТИЧЕСКИХ ДЕЙСТВИЙ

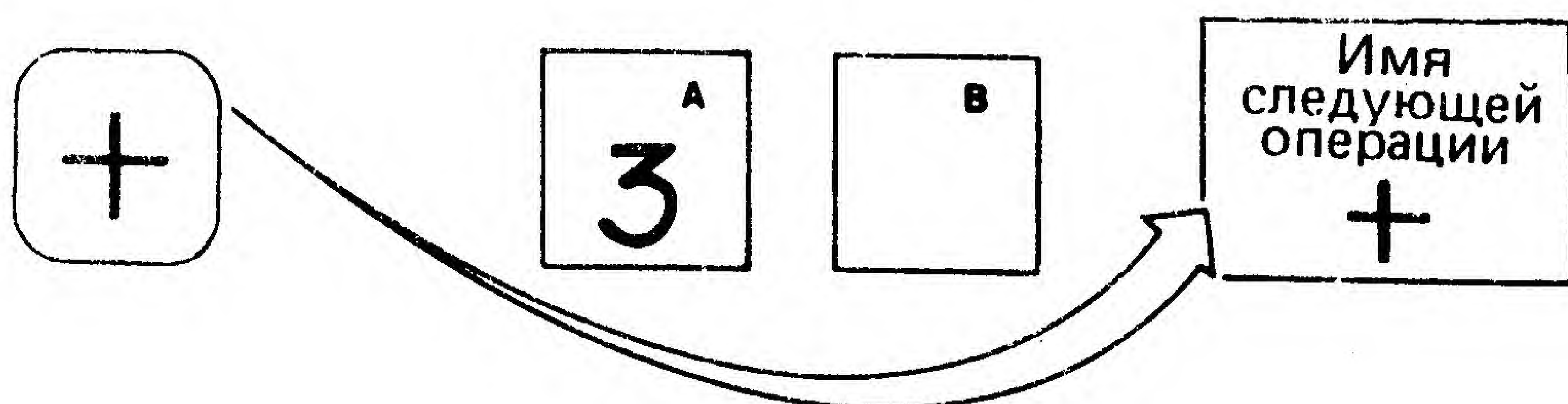
Компьютер вряд ли удовлетворит ваши запросы, если не сможет выполнять арифметические действия. Если вы никогда ранее не имели дело с компьютером, то вам может показаться странным, что он (или даже карманный калькулятор) вообще производит какие-то вычисления. Мы не можем в рамках данной книги

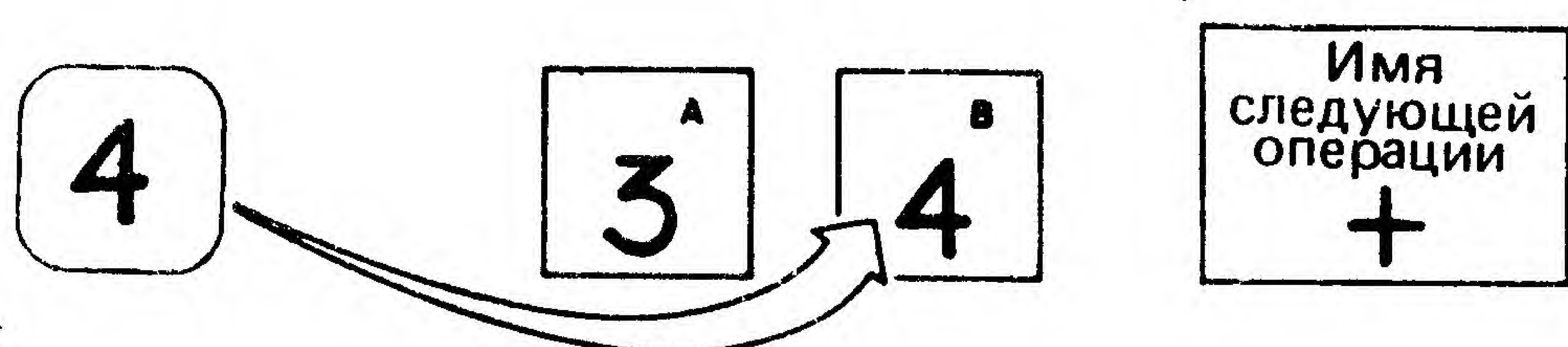
подробно описать весь вычислительный процесс, но поверьте, что чуда здесь нет никакого.

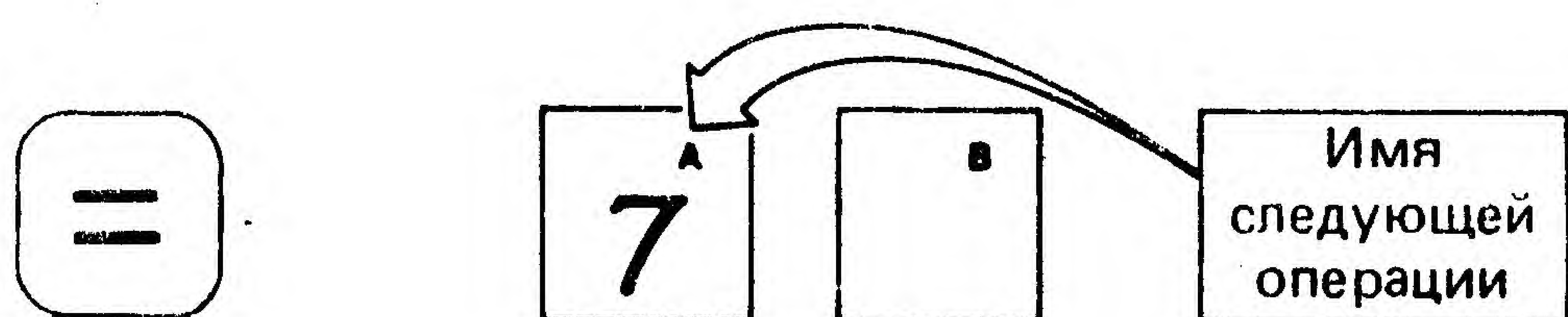
В принципе компьютеры при выполнении операций разбивают их на очень мелкие этапы и реализует предельно элементарные действия. Для нас с вами операция сложения «3 + 4» не составляет труда и вы, не задумываясь, называете результат: «7», для компьютера же это очень длинная цепочка действий. Представьте себе, что у вас есть карманный калькулятор и вы хотите выполнить на нем такое сложение. Чтобы получить результат, требуется нажать клавиши в указанном ниже порядке.



Когда вы поочередно нажимаете клавиши, как показано на рисунке, происходит следующее:



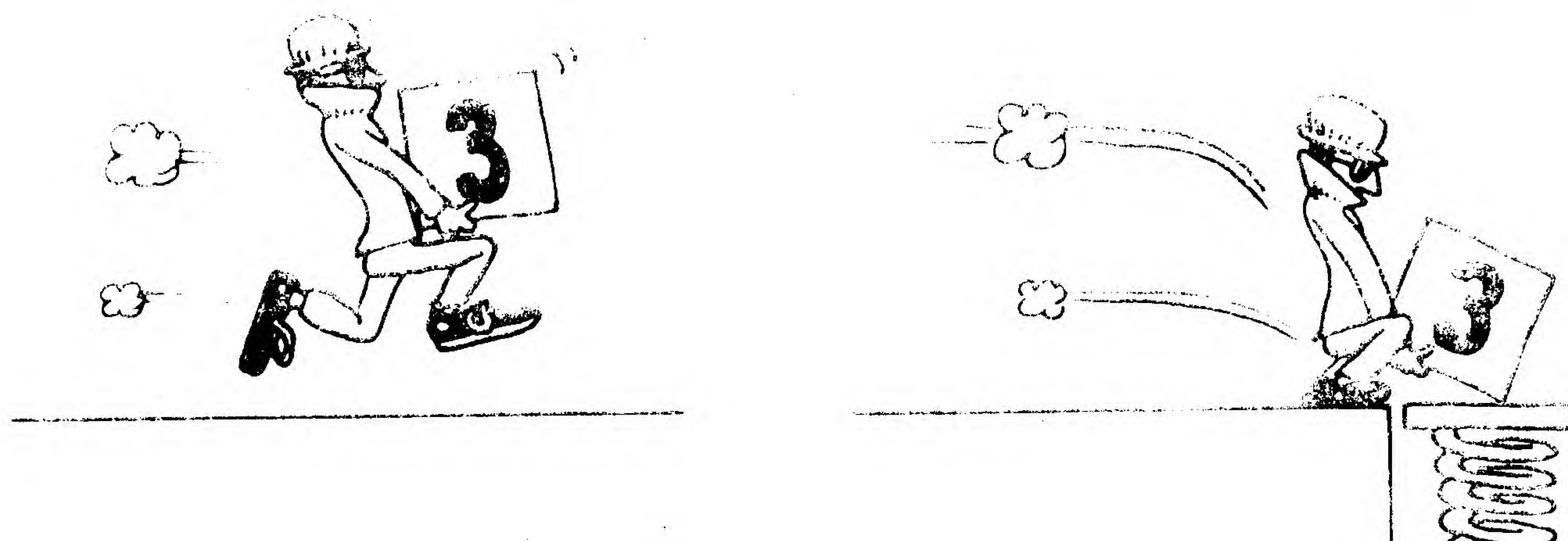




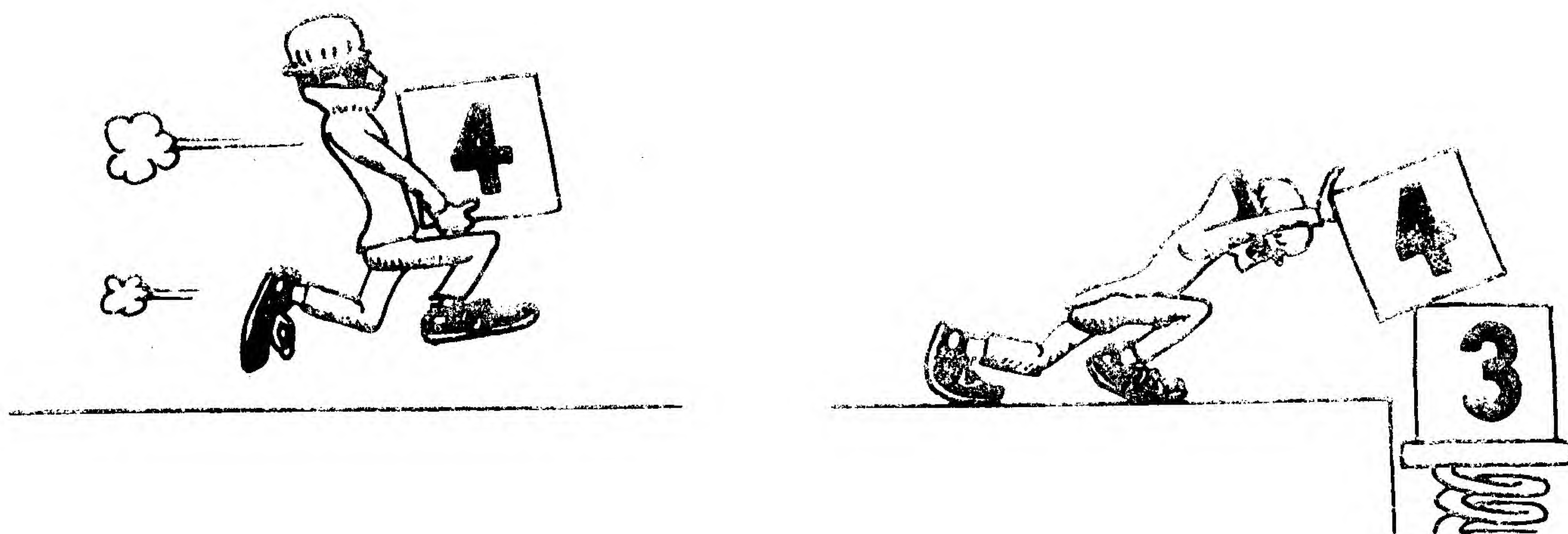
Многие калькуляторы и компьютеры решают арифметические задачи аналогичным способом. Вы можете этого и не знать, но компьютер действительно запоминает числа в различных участках своей оперативной памяти и затем выполняет над ними необходимые операции. Центральный участок памяти, где временно за-

поминаются числа, над которыми затем будет выполнена операция, называется *стеком*. Числа «вталкиваются в стек» и *впоследствии* операции выполняются именно над данными числами[1]. Поясним это на примере. Введите со своего терминала строку

```
3 4 + .<return> 7 ok
```



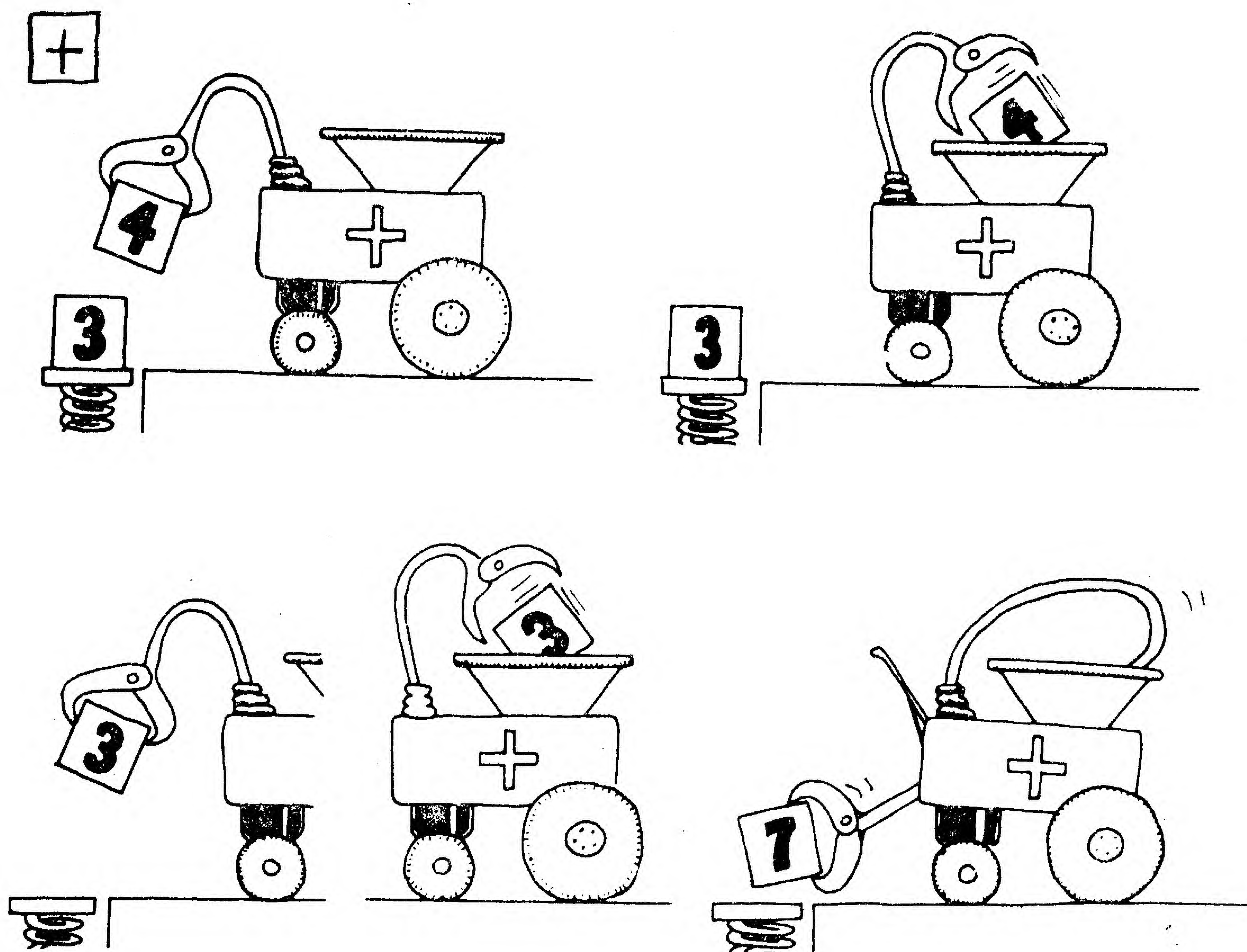
Вспомните, что при вводе числа интерпретатор текста переправляет его обработчику чисел **NUMBER**, который помещает это число в некоторый участок памяти. Таким участком является стек.
Короче говоря, когда вы вводите число 3 с терминала, вы «вталкиваете» его в стек.



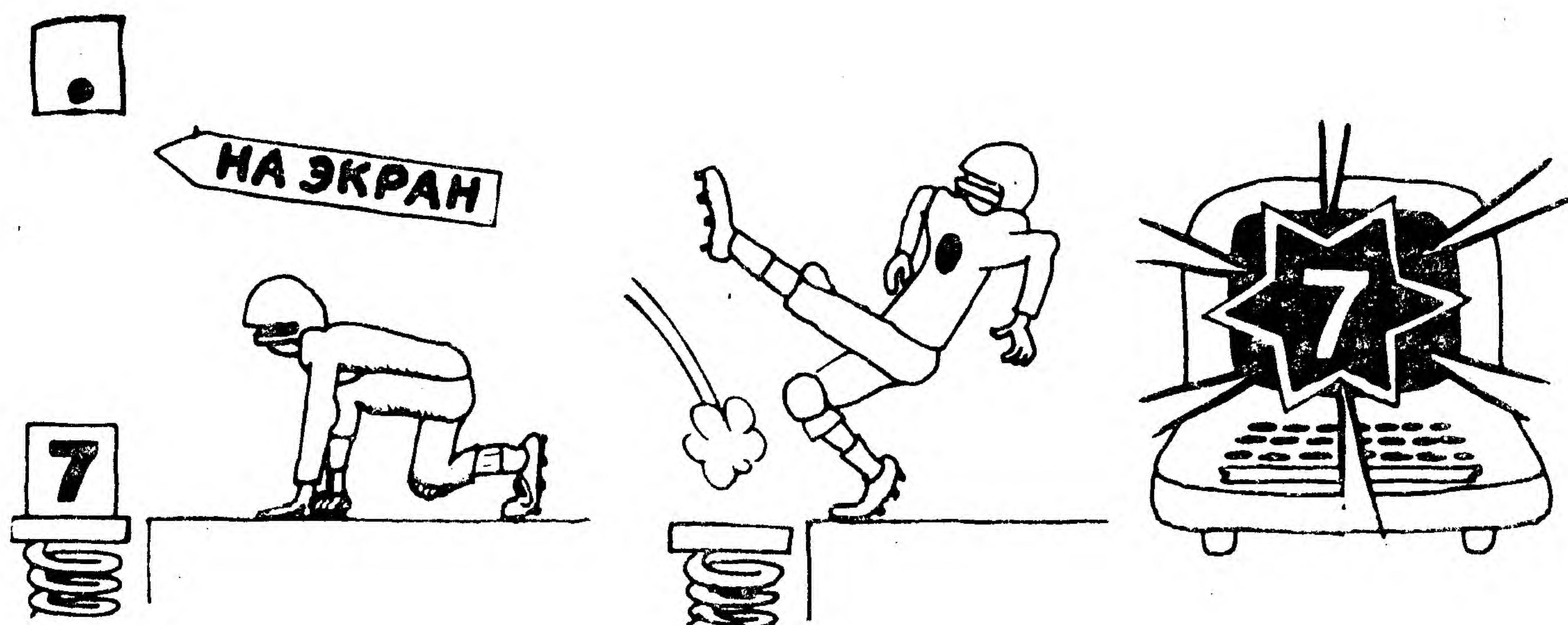
Далее в вершину стека помещается число 4, «проталкивая» тройку внутрь.

---

[1] *Для пользователей карманных калькуляторов.* В обычных калькуляторах, как правило, применяется и стек, и постфиксная запись арифметики.



Следующее слово из входного потока *должно быть* найдено в словаре. Знак + был предварительно определен как последовательность таких действий: «взять два верхних числа из стека, просуммировать их и поместить результат обратно в стек».



Слово . также находится в словаре. Оно входит в состав ядра Форт-системы и определено таким образом: «взять число из стека и вывести его на терминал».

ПОСТФИКСНАЯ ЗАПИСЬ

«Но, позвольте, — скажете вы, — почему на Форте нужно писать

```
3 4 +
```

а не

```
3 + 4
```

как это принято?» Дело в том, что в Форте применяется *постфиксная* запись (называемая так потому, что знак операции располагается после чисел) вместо *инфиксной* (где знак операции располагается *между* числами), поэтому все слова, которым «требуются» числа, могут их взять из стека. Например, слово + берет из стека два числа и складывает их, слово . берет одно число и выводит его, **SPACES** берет одно число и выводит соответствующее ему количество пробелов, **EMIT** берет число, которое изображает какой-либо символ, и выводит этот символ. Даже слово STARS, которое мы определили сами, берет число из стека и выводит заданное этим числом количество звездочек.

Когда *все* операции, которые должны быть выполнены над числами, уже находящимися в стеке, определены, обеспечить взаимодействие между ними достаточно просто, даже если программа становится сложной.

Ранее отмечалось, что Форт позволяет использовать слово любым из двух способов: либо называя его, либо применяя в определении другого слова, тем самым именуя то *другое* слово. Постфиксная запись является частью механизма, который предоставляет вам такую возможность. Например, предположим, что вам требуется слово, которое всегда прибавляет число 4 к любому числу, находящемуся в стеке. Назовем это слово

```
ПЛЮС-ЧЕТЫРЕ
```

что означает «больше на четыре», и определим его следующим образом:

```
: ПЛЮС-ЧЕТЫРЕ   4 + ;<return>
```

Выполним проверки

```
3 ПЛЮС-ЧЕТЫРЕ .<return> 7 ok
```

и еще раз

```
-10 ПЛЮС-ЧЕТЫРЕ .<return> -6 ok
```

Число 4, находящееся *внутри* определения, помещается в стек таким же образом, как если бы оно находилось вне определения. Затем слово + складывает два числа, хранящихся в стеке. Так как операция сложения всегда выполняется над содержимым стека, тот факт, что число 4 пришло из определения, а 3 нет, не имеет значения.

В дальнейшем мы перейдем к более сложным примерам, и тогда процесс занесения значений в стек и арифметика в постфиксной записи будут вам намного понятнее. Чем больше операций вовлекается в процесс, тем важнее становится вопрос взаимодействия между ними. Применение стека упрощает это взаимодействие.

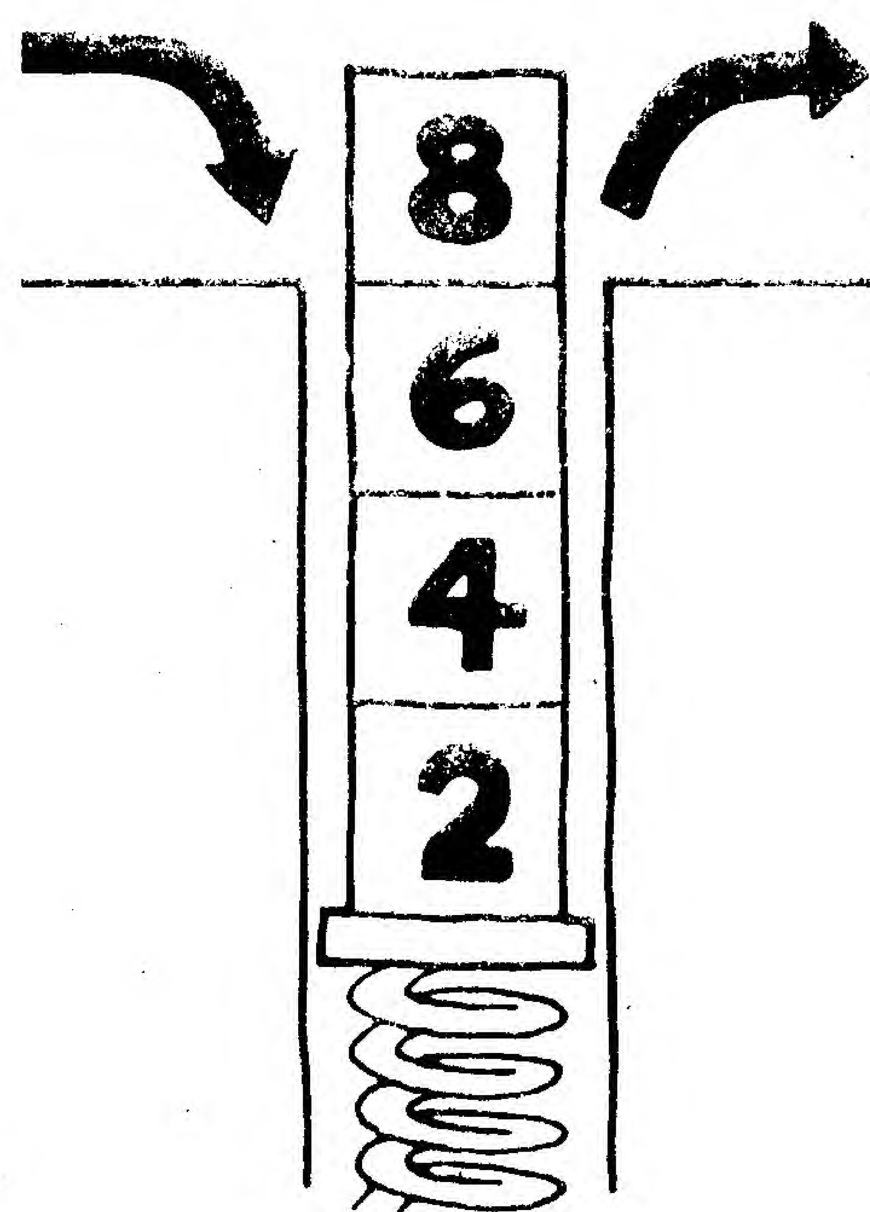
РАБОТА СО СТЕКОМ

Мы уже продемонстрировали некоторые методы работы со стеком и постфиксную запись. Теперь рассмотрим этот вопрос более детально.

Стек Форта функционирует по принципу «последним занесен — первым выбран» (LIFO). Вы это видели в приведенном выше примере. Число 3 было занесено в стек первым, а затем в него занесли число 4, «протолкнув» вглубь стека тройку. Позднее при выполнении операции сложения машина выбрала первым число 4, так как оно находилось в вершине стека.

В качестве еще одного примера выполним другую операцию . . Вспомните, что каждое слово . берет одно значение из стека и выводит его на экран. Четыре точки, следовательно, возьмут четыре числа:

```
2 4 6 8 . . . .<return> 8 6 4 2 ok
```



Система читает символы из входного потока слева направо. При вводе крайнее правое значение на экране дисплея будет находиться последним в стеке, т. е. в его *вершине*. При выводе крайнее правое значение поступает на экран с самого *дна* стека.

Будьте внимательны, чтобы не допустить ошибки. Наберите на клавиатуре

```
10 20 30 . . . .
```

(*четыре точки*) и нажмите клавишу RETURN. В результате вы получите[1]:

```
10 20 30 . . . .<return> 30 20 10 . Стек пуст
```

Каждая точка удаляет из стека одно значение. Четвертая точка «обнаружит», что в стеке нечего взять для вывода на дисплей, о чем вы и получите сообщение.



Такая ошибка называется потерей элемента стека. (Заметьте, что потеря элемента стека вовсе *не* «ok».) В противном случае, когда вы исчерпали емкость стека, возникает ситуация его переполнения. Однако стек настолько глубок, что такое событие практически нереально, если только вы не совершите что-то из ряда вон выходящее.

Как правило, вас не должно интересовать содержимое всего стека — вам нужны только те числа, с которыми вы работаете в данный момент. Остальные же значения, если таковые были занесены в стек ранее, должны в сохранности находиться в стеке до того момента, пока они не понадобятся. Если вы, к примеру, захотите нарисовать прямоугольник, применяя ту же технику, что и

[1] *Для любознательных.* На самом деле на экран выводится всегда то значение, которое находится в вершине стека. Поэтому если в стеке ничего нет, выводится следующее значение, находящееся глубже последнего, а именно нуль. И только тогда обнаруживается ошибка. Ошибочное слово (в нашем случае точка) выводится на экран, а за ним — сообщение об ошибке.

при создании буквы F, то для начертания сторон можно написать слово

```
: СТОРОНЫ   STAR  5 SPACES  STAR ;
```

которое при выполнении дает вам следующее:

```
*     *
```

Предположим, вы хотите научиться рисовать прямоугольник *любого* размера. Тогда вы должны не заключать число 5 в *определении*, а передавать его слову СТОРОНЫ как *аргумент*. В этом случае ваше определение будет выглядеть так:

```
: СТОРОНЫ   STAR  SPACES  STAR ;
```

Теперь нужно обращаться к слову СТОРОНЫ, скажем, таким образом:

```
5 СТОРОНЫ
```

При этом несмотря на то, что при первом употреблении слова STAR в стек заносится число 42, предназначенное для **EMIT**, заданное вами число 5 будет благополучно дожидаться в стеке выполнения своего слова **SPACES**.

СТЕКОВАЯ НОТАЦИЯ

Передачу аргументов определениям легко и удобно производить через стек[1]. Но при пользовании стеком вся ответственность за соответствие его состояния так называемой «стековой нотации» ложится на вас. Иными словами, при определении некоторого слова вы должны представлять себе, какие аргументы это слово при своем выполнении выбирает из стека, а какие в нем оставляет или «возвращает», и при выполнении слова убедиться в том, что в действительности так и происходит.

Применительно к нашему простому примеру ПЛЮС-ЧЕТЫРЕ изложенное выше означает, что перед выполнением указанного слова в стеке должен находиться один аргумент, а после его вы-

---

[1] *Для знатоков семантики.* В математике слово *аргумент* ассоциируется с независимой переменной некоторой функции. Компьютерные лингвисты заимствовали этот термин для обозначения значений, передаваемых функциям или подпрограммам. Кроме того, для описания аналогичных объектов используется слово *параметры.*

полнения оставаться результат в виде одного значения. Если вы поместите внутри определения ПЛЮС-ЧЕТЫРЕ «точку» для вывода получаемого результата на дисплей, то изменится стековый эффект: данное слово не должно будет возвращать в стек значение, полученное в результате своего выполнения.

Чтобы иметь возможность визуально согласовывать стековые эффекты при выполнении различных слов, программисты применяют специальную запись — *стековую нотацию*. Такой комментарий обязательно должен присутствовать в программных листингах и глоссариях (вид документации, в которой содержатся перечни слов, используемых в вашей прикладной программе). Прежде чем показать вам, как выглядит стековая нотация, обсудим вопрос о том, что представляют собой комментарии в Форте.

Комментарий — это некоторая информация, предназначенная только для человека: она ни выполняется, ни компилируется. В Форте слово **(** , *левая круглая скобка,* символизирует начало комментария. Текстовому интерпретатору предписывается осуществлять пропуск последующего текста до тех пор, пока не встретится правая круглая скобка, означающая конец комментария. Поскольку **(** является словом, вы обязаны отделять этот символ пробелом так же, как и в случае применения слова **."**[1].



Вы можете использовать обычный комментарий внутри некоторого определения следующим образом:

```
: НИЧЕГО  ( это слово ничего не выполняет )  ;
```

Текст «это слово ничего не выполняет» является комментарием.

Вернемся к стековой нотации. Основной формат комментария этого вида выглядит так:

```
( -- )
```

[1] *Для начинающих.* Закрывающая круглая скобка не является словом. Это просто символ, который служит ограничителем для слова **(** . (Вспомните, что для слова **."** ограничителем является символ **"**.)

Такой комментарий означает, что данное определение никакого эффекта на стек не оказывает. К словам подобного рода относится **CR** или специфицированное нами слово STAR. (Во время своего исполнения слово STAR помещает в стек число 42, но извлекает его из стека до завершения работы, так что стековых эффектов в рассматриваемом случае нет.) Принято отделять стековую нотацию от имени определения двумя пробелами:

```
: STAR  ( -- )  42 EMIT ;
```

Напоминаем, что стековая нотация ничего не значит для Форта, но очень помогает программисту, пытающемуся разобраться в конкретной программе.

Если некоторое слово *должно выбрать* из стека аргументы, то эти аргументы перечисляются *слева* от двойного дефиса. Например, стековая нотация для слова . («точка») выглядит следующим образом:

```
( n -- )
```

(Буква n заменяет число.) Если слово *возвращает* стеку аргументы, то они перечисляются *справа* от двойного дефиса. Стековая нотация для слова + имеет вид:

```
( n1 n2 -- сумма )
```

Для обозначения аргументов вы можете использовать имена, сокращения или просто нумеровать их: n1, n2, n3 и т. д., как было сделано в приведенном выше примере.

Когда вы указываете несколько аргументов справа или слева от дефиса, необходимо строго соблюдать порядок их размещения. Запомните следующее правило: *крайний правый* объект в стековой нотации является *верхним* элементом стека.

(n1 n2 --*сумма*)



Этот элемент верхний

Это легко запоминается, поскольку перечисление аргументов в стековой нотации совпадает с порядком их ввода. Если вы вво-

дите в стек для некоторого слова числа 1 2 3, то стековый комментарий будет таков:

```
( 1 2 3 -- )
```

т. е. 1 окажется на дне, а 3 — в вершине стека.

Так как вы, очевидно, уже разобрались в правилах записи, в дальнейшем будет опускаться <return>, за исключением тех случаев, где это необходимо для ясности. Ответы компьютера в книге всегда подчеркнуты, поэтому вам должно быть понятно, когда следует нажимать клавишу RETURN.

Ниже приводится список слов Форта, которые вам уже знакомы, вместе с их стековой нотацией (n замещается числом, c — символом).

: xxx yyy ;	(--)	Определение нового слова с именем xxx, состоящее из слова или слов yyy.
CR	(--)	Возврат каретки и перевод строки.
SPACES	(n --)	Вывод заданного числа пробелов.
SPACE	(--)	Вывод одного пробела.
EMIT	(c --)	Вывод символа.
." xxx"	(--)	Вывод строки символов xxx. Символ " является признаком конца строки.
+	(n1 n2 -- сумма)	Суммирование.
	(n --)	Вывод числа, за которым следует один пробел.
(xxx)	(--)	Комментарий, который текстовым интерпретатором не воспринимается. Символ) является ограничителем.

В следующем разделе мы расскажем вам о том, как заставить компьютер выполнять более сложные арифметические операции.

ОСНОВНЫЕ ТЕРМИНЫ

Выполнение. Применительно к слову — это выполнение операций, заданных в скомпилированном определении данного слова.

Входной поток. Текст, который должен быть прочитан текстовым интерпретатором. Это может быть текст, только что набранный на терминале или ранее записанный на диск.

Глоссарий. Список слов, определенных в Форте, с их стековыми нотациями (какого рода информация помещается в стек перед выполнением слова и что остается в стеке в качестве результата).

Интерпретация. Чтение из входного потока при обращении к интерпретатору текста и поиск каждого встретившегося слова в словаре. В случае неудачи это слово трактуется как число.

Инфиксная запись. Метод записи: знак операции располагается между операндами, над которыми она выполняется, например 2 + 5.

Компиляция. Генерация по исходному тексту элемента словаря (внутренняя форма определения). Следует отличать от EXECUTE.

Переполнение стека. Аварийная ситуация, которая создается в том случае, когда вся область памяти, отведенная под стек, заполнена данными.

Постфиксная запись. Метод записи: знак операции следует за операндами, над которыми эта операция выполняется, например 2 5 +. Также известен как обратная польская запись.

Потеря элемента стека. Аварийная ситуация, которая возникает в том случае, когда для выполнения какой-либо операции требуется элемент из стека, а стек пуст.

Расширяемость. Характеристика языка, означающая, что программист может добавлять новые средства или модифицировать существующие.

Словарь. В Форте это перечень слов и определений, включающий в себя как «системные» определения (созданные при генерации системы), так и «пользовательские» (которые создаете вы сами). Словарь размещается в памяти компьютера в компилируемой форме.

Слово. В Форте это имя определения.

Стек. Участок памяти, в который данные помещаются и из которого они удаляются по принципу «последним пришел — первым обслужен» (LIFO).

LIFO. Принцип функционирования стека (последним пришел — первым обслужен). Так, коробка для хранения теннисных мячей имеет структуру LIFO: последний помещенный в нее мяч вы выбираете первым.

УПРАЖНЕНИЯ

Прежде чем вы приступите к решению задач, запомните следующее простое правило: каждое из перечисленных ниже слов должно завершаться соответствующим символом:

Слово	*Завершающий символ*
:	;
."	"
(	)

1.1. Определите слово с именем ДАР, которое при своем выполнении выдаст название какого-то подарка. Например, вы можете определить

```
: ДАР   ." подставку для книг " ;
```

Теперь определите слово с именем ДАРИТЕЛЬ, которое выводит на печать имя некоего лица, а затем слово с именем СПАСИБО, определение которого включает вновь созданные слова ДАР и ДАРИТЕЛЬ и выводит на печать сообщение, аналогичное следующему:

```
Дорогая Маша,
    спасибо за подставку для книг. ok
```

1.2. Определите слово с именем МИНУС-ДЕСЯТЬ, которое выбирает некоторое число из стека, вычитает 10 и помещает полученный результат в стек. (*Подсказка*: вы можете использовать слово +.) Не забудьте включить в определение стековую нотацию.

1.3. После того как вы введете слова при решении упр. 1.1, переопределите слово ДАРИТЕЛЬ так, чтобы можно было вывести на печать чье-то другое имя (не переопределяя слово СПАСИБО), выполните последнее снова. Сможете ли вы объяснить, почему слово СПАСИБО выводит на печать имя первого дарителя?

Глава 2

ВЫПОЛНЕНИЕ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ

Прежде чем продолжить изучение Форта, обсудим некоторые специфические вопросы, связанные с выполнением арифметических операций. В частности, в дополнение к суммированию введем несколько других арифметических операций, а также рассмотрим операторы для переупорядочения значений, находящихся в стеке, после чего вы сможете записывать на Форте математические выражения.

РЕЖИМ КАЛЬКУЛЯТОРА

Ниже приводятся четыре простейшие операции над целочисленными значениями, записанные на языке Форт[1]:

```
+   плюс        ( n1 n2 -- сумма )          сложение  ( n1 + n2 )
-   минус       ( n1 n2 -- разность )       вычитание ( n1 - n2 )
*   звездочка   ( n1 n2 -- произведение )   умножение ( n1 * n2 )
/   слэш        ( n1 n2 -- частное )        деление   ( n1 / n2 )
```

В отличие от калькулятора на терминале компьютера не предусмотрены специальные клавиши для выполнения операций умножения и деления. Вместо них мы пользуемся клавишами * и /.

Из предыдущего раздела вы уже знаете, что можно сложить два числа, поместив их в стек и выполнив слово +, а затем ., чтобы вывести результат на терминал:

---

[1] *Для нематематиков.* Хотя данная глава и напоминает немного учебник по алгебре, решение математических задач — всего лишь небольшая часть из того, что вы сможете делать с помощью Форта. Позднее вы познакомитесь с другими применениями Форта. Здесь же уместно напомнить, что целые числа — это такие круглые числа, как ... —3, —2, —1, 0, 1, 2, 3, ..., а целочисленная арифметика (что достаточно логично) — операции над целыми числами.

17 5 + . 22 ok

Вы можете выполнить таким образом все арифметические операции даже без составления «программы», используя Форт-систему как калькулятор. Решите задачу на умножение:

7 8 * . 56 ok

Как видите, знак операции следует за значениями. Если же вы производите вычитание и деление, необходимо учитывать *порядок следования значений* («7 — 4» не эквивалентно «4 — 7»).

Запомните следующее правило: для записи выражения в постфиксной форме достаточно передвинуть знак операции в конец этого выражения:

инфиксная запись	постфиксная запись
3 + 4	3 4 +
500 - 300	500 300 -
6 X 5	6 5 *
20 / 4	20 4 /

Поэтому чтобы выполнить вычитание

7 − 4 =

наберите на клавиатуре

7 4 − . 3 ok

Для начинающих, которым нравится развлекаться за терминалом. Если вы из тех, кто любит постигать суть вещей, не читая руководства, вы неизбежно столкнетесь с рядом проблем. Во-первых, как уже отмечалось, описанные выше операции являются *целочисленными операциями*. Это означает не только то, что вы не имеете права их выполнять над дробными числами, например

10.00 2.25 +

но и то, что вы можете получить лишь целочисленный результат, т. е.

21 4 / . 5 ok , а не 5.25 ok

Во-вторых, если вы попытаетесь выполнить умножение:

10000 10 *

или перемножить подобные большие числа, то получите неожиданный результат. Поэтому мы вас предупреждаем, что все операции, о которых шла речь выше, а также операция, для вывода результата могут выполняться только над числами, лежащими в диапазоне от — 32 768 до 32 767. Эти числа нызываются *числами одинарной длины со знаком.*

+32767 0 –32768	Допустимый диапазон чисел одинарной длины со знаком

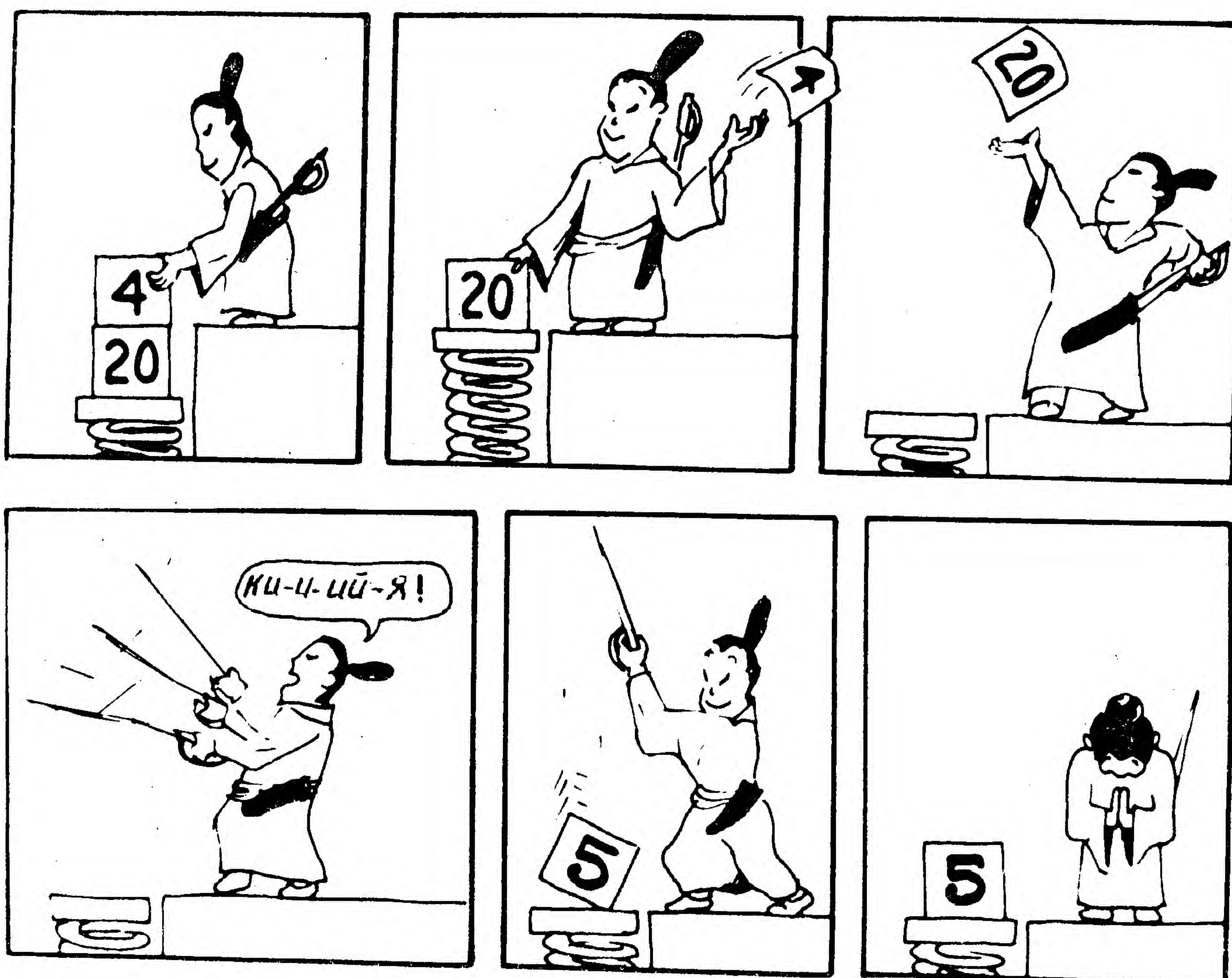
Напомним, что, рассматривая перечень слов Форта, мы употребляли букву n, чтобы обозначить место, где должно находиться число. Так как в Форте числа одинарной длины используются гораздо чаще чисел других типов, вместо n следует подставлять число одинарной длины. Конечно, существуют операции, которые выполняются и над значениями из расширенного диапазона (двойной длины). Они обозначаются буквой d.

Все эти непонятные пока проблемы будут объяснены в свое время, так что не снижайте внимания.

Порядок чисел остается тем же. В качестве примера решите задачу на деление:

```
20 4 / . 5 ok
```

Слово **/** определено таким образом, что нижнее число в стеке делится на число, находящееся в его вершине.

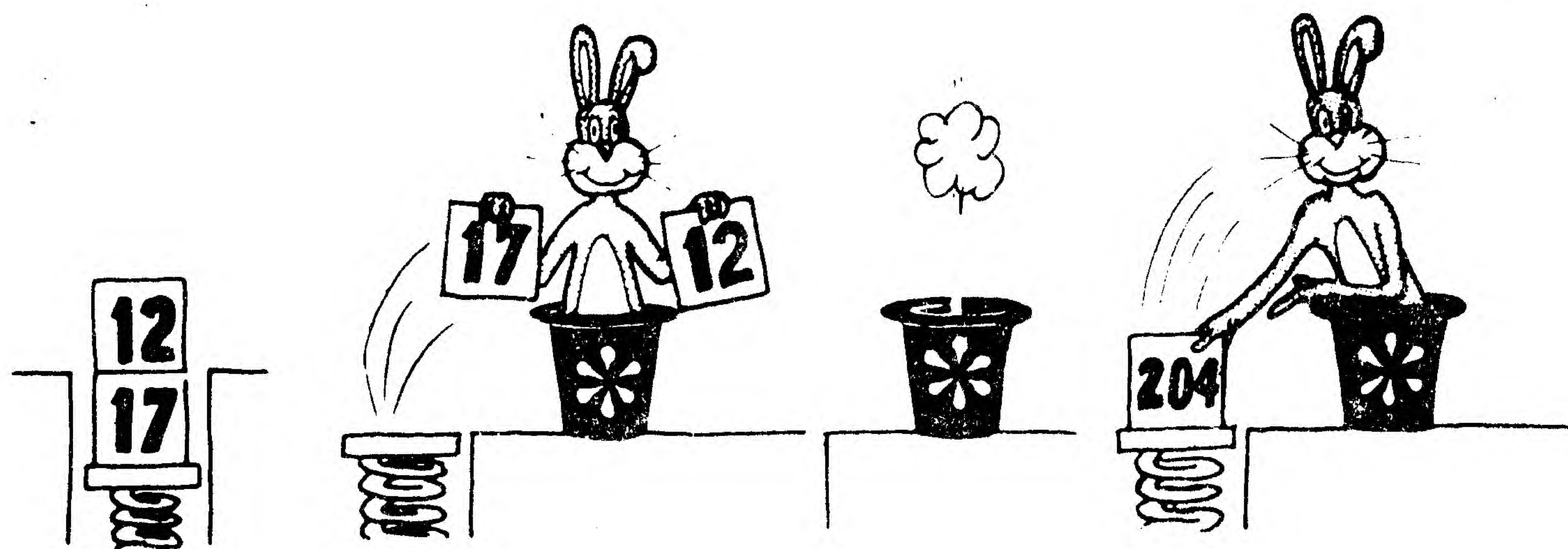


Как поступить, если необходимо выполнить несколько операций? Например:

```
4 + (17 * 12)
```

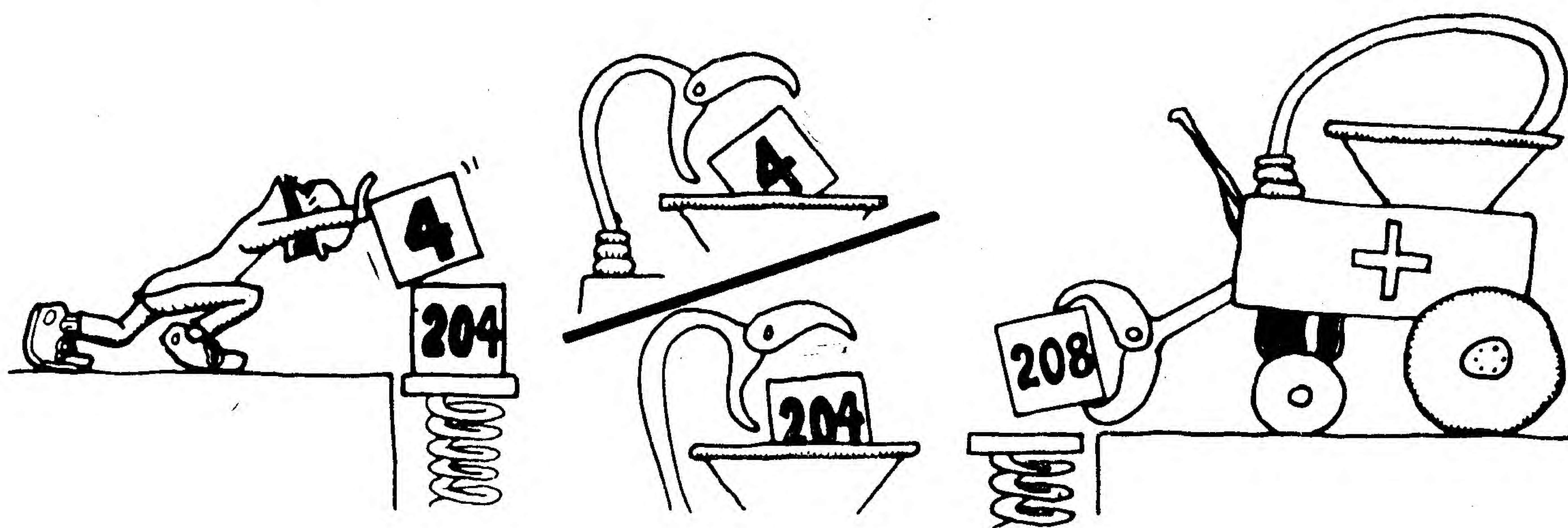
Как известно, сначала нужно выполнить операцию, указанную в скобках, т. е. 17 умножить на 12, а *затем* добавить четыре. На Форте это будет выглядеть так:

```
17 12 * 4 + . 208 ok
```



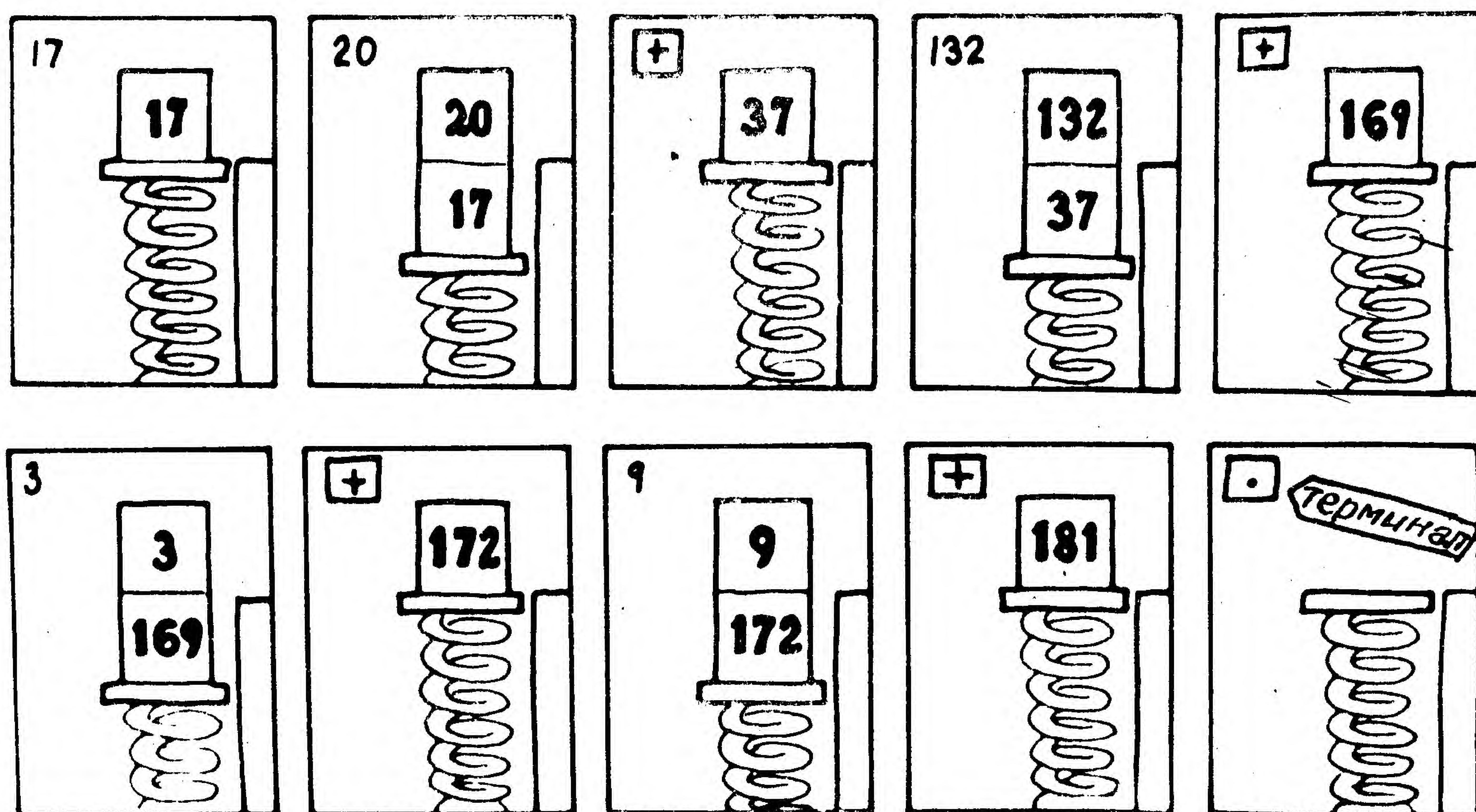
Числа 17 и 12 помещаются в стек. Слово * перемножает их и возвращает результат в стек.

Далее число 4 помещается в стек над числом 204. Слово + «выкатывает» суммирующую машину и складывает эти два числа, а в стек возвращается только результат.



Предположим, вы хотите сложить пять чисел. Вы можете это сделать на Форте, скажем, так:

```
17 20 + 132 + 3 + 9 +  . 181 ok
```



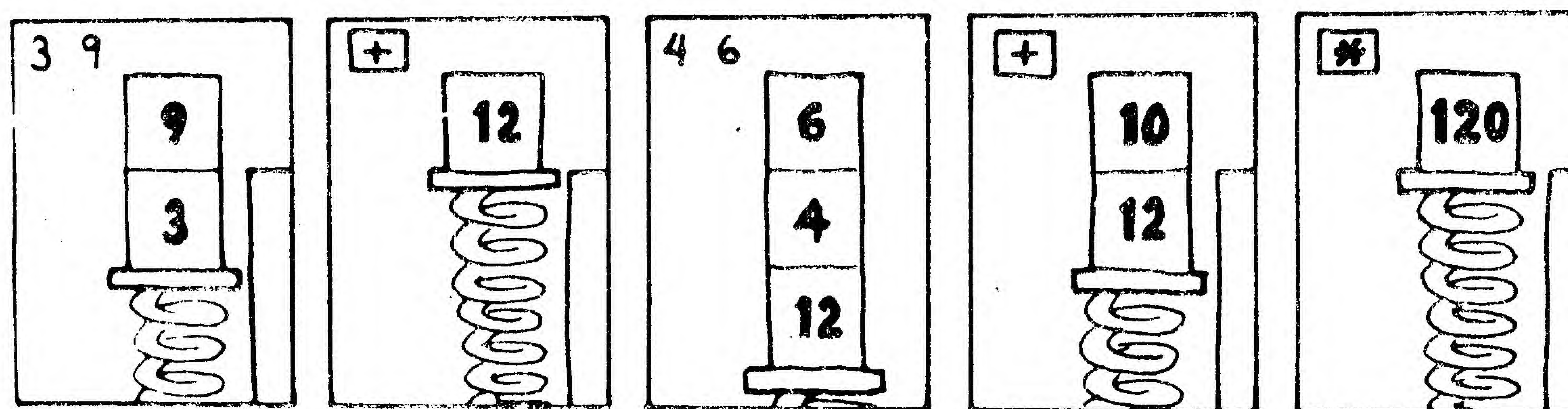
Еще одна интересная задача:

```
(3+9) * (4+6)
```

Чтобы ее решить, мы должны сначала сложить числа 3 и 9, затем 4 и 6 и, наконец, перемножить полученные две суммы. На Форте это можно записать следующим образом:

```
3 9 +  4 6 +  *  . 120 ok
```

В результате вы получите.



Заметьте, что мы весьма кстати сохранили сумму, равную 12, в стеке на то время, пока складывали числа 4 и 6.

Помните, что мы еще не пишем определений, а лишь используем Форт-систему в режиме калькулятора. Начинающим, возможно, потребуется решить несколько практических задач для того, чтобы чувствовать себя увереннее в применении постфиксной записи.

ПРАКТИЧЕСКИЕ ЗАДАЧИ НА ПРИМЕНЕНИЕ ПОСТФИКСНОЙ ЗАПИСИ (УПРАЖНЕНИЕ 2-А)

Для начала представьте выражения из левого столбца в постфиксной записи, используя только карандаш и бумагу. Например, если дано

ab + c	2 · 3 + 4	(10)

У вас должно получиться следующее:

```
a b *  c +
```

Затем проверьте полученные выражения, подставляя в них числа из среднего столбца и применяя Форт в режиме калькулятора. Правильный результат приведен в правом столбце. В нашем случае

```
2 3 *  4 + . 10 ok
```

1.	c(a + b)	3(4 + 5)	(27)
2.	$\frac{ab}{100}$	$\frac{80 \cdot 90}{100}$	(72)
3.	$\frac{3a - b}{4} + c$	$\frac{(3 \cdot 9) - 7}{4} + 2$	(7)
4.	$\frac{a + 1}{4}$	$\frac{7 + 1}{4}$	(2)
5.	x(7x + 5)	10 ((7 10) + 5)	(750)

Преобразуйте следующие выражения из постфиксной формы в инфиксную:

```
6.  a b -  b a +  /

7.  a b 10 *  /
```

Ответы к упражнению 2-А

```
1.  a b +  c *        или
    c a b + *

2.  a b *  100 /

3.  3 a *  b -  4 /  c +

4.  a 1 +  4 /

5.  7 x *  5 +  x *

    a - b
6.  -----
    b + a

     a
7.  ---
    10b
```

РЕЖИМ ОПРЕДЕЛЕНИЙ

Как вы уже знаете из первой главы, новые слова можно определять с помощью чисел и ранее определенных слов. Теперь, используя некоторые из изученных нами арифметических операций, рассмотрим другие возможности языка.

Допустим, вы хотите перевести различные единицы измерения в дюймы. Вам известно, что 1 ярд = 36 дюймам и 1 фут = 12 дюймам[1]. Поэтому вы можете определить эти два слова следующим образом:

```
: ЯРД>ДМ   ( ярды -- дюймы )   36 * ; ok
: ФУТ>ДМ   ( футы -- дюймы )   12 * ; ok
```

Здесь определяемые имена означают «ярды-в-дюймы» и «футы-в-дюймы» соответственно. Ниже приводятся примеры выполнения указанных слов:

```
10 ЯРД>ДМ . 360 ok
 2 ФУТ>ДМ . 24 ok
```

[1] 1 дюйм = 2,54 см, 1 фут = 30,48 см, 1 ярд = 0,914 м. — *Примеч. пер.*

Если результат должен *всегда* выражаться в дюймах, то необходимо ввести следующие определения:

```
: ЯРДОВ   ( ярды -- дюймы )   36 * ; ok
: ФУТОВ   ( футы -- дюймы )   12 * ; ok
: ДЮЙМОВ   ( -- ) ; ok
```

Таким образом, можно записать выражение:

```
10 ЯРДОВ 5 ФУТОВ + 9 ДЮЙМОВ + . 429 ok
```

Отметим, что назначение слова ДЮЙМОВ — напомнит вам, для чего здесь поставлено число 9. Если вы хотите писать грамотно, то должны добавить три определения:

```
: ЯРД    ЯРДОВ ; ok
: ФУТ    ФУТОВ ; ok
: ДЮЙМ ; ok
```

Набирая при вводе эти существительные в единственном числе, вы получите такой же результат:

```
1 ЯРД 5 ФУТОВ + 1 ДЮЙМ + . 97 ok
6 ЯРДОВ 1 ФУТ + . 228 ok
```

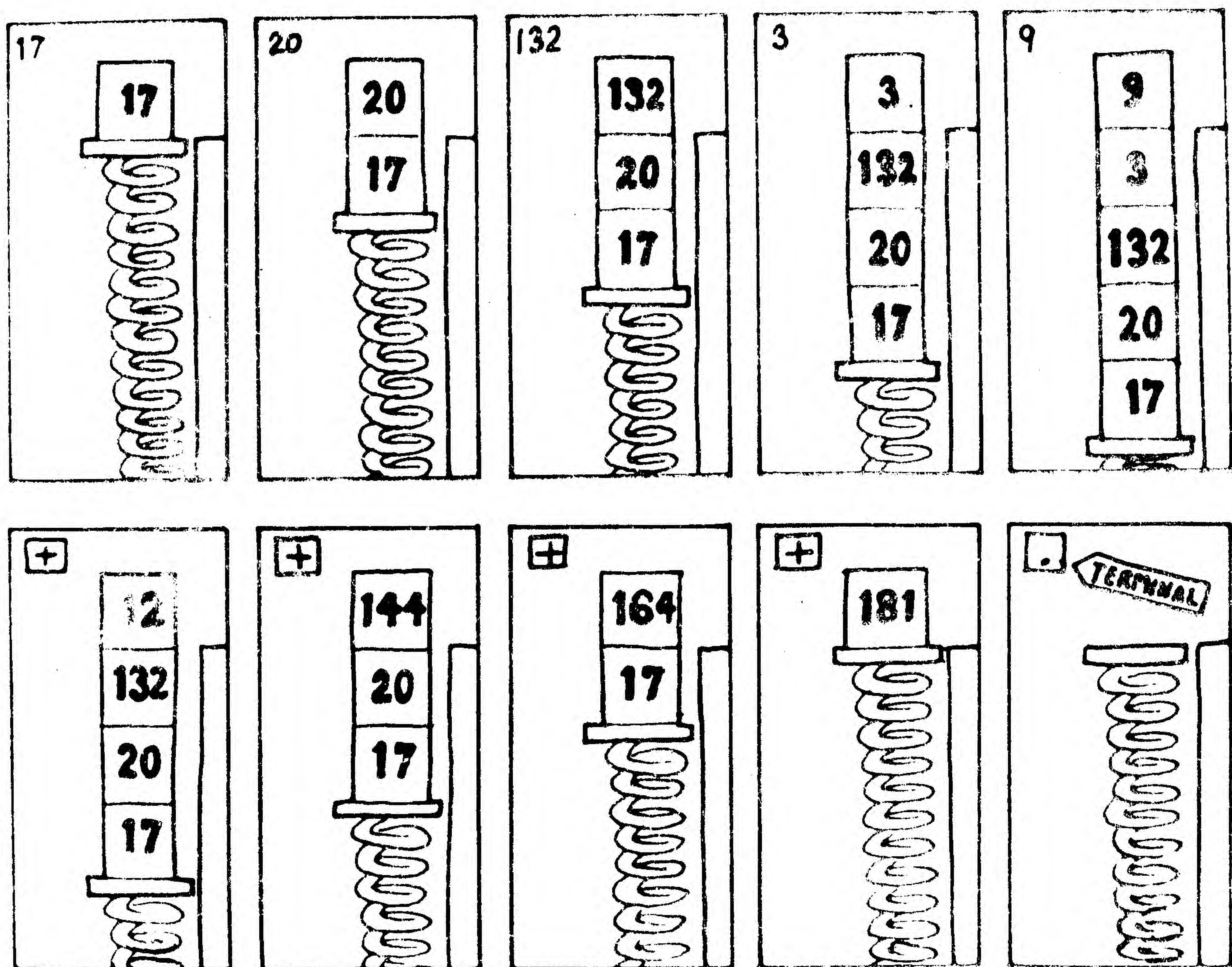
До сих пор мы вводили слова, определения которых содержали только одну арифметическую операцию. Но с тем же успехом вы можете при необходимости использовать внутри любого определения несколько операций.

Допустим, вы хотите получить слово, которое суммирует пять чисел, находящихся в стеке. Ранее мы уже суммировали пять чисел следующим образом:

```
17 20 + 132 + 3 + 9 + . 181 ok
```

однако можно набрать на клавиатуре и такую строку:

```
17 20 132 3 9 + + + + . 181 ok
```



Мы получили тот же ответ, несмотря на то что собрали все числа в одну группу, а все знаки операций — в другую. Запишем наше определение так:

```
: 5#СУММА  ( n1 n2 n3 n4 n5 -- сумма )  + + + + ; ok
```

и выполним слово, как показано ниже:

```
17 20 132 3 9 5#СУММА . 181 ok
```

Ниже приводится еще одно выражение, для которого нужно записать определение[1]:

```
(a + b)  * c
```

---

[1] *Для начинающих, которым нравятся задачи, сформулированные словесно.* Реактивный самолет летит со средней скоростью 600 миль/ч. Скорость попутного ветра составляет 25 миль/ч. Сколько миль пролетит этот самолет при таких условиях за пять часов?
Если мы определим слово

```
: РАССТОЯНИЕ  ( время скорость попутный-ветер -- расстояние )
     + * ;
```

то можно затем ввести следующее:

```
5 600 25 РАССТОЯНИЕ . 3125 ok
```

Попытайтесь выполнить это слово с различными значениями, учитывая и встречный ветер (отрицательные значения).

Как вы видели из упр. 2-А, это выражение может быть записано в постфиксной форме:

```
c a b + *
```

Итак, мы можем записать наше определение:

```
: РЕШЕНИЕ  ( c a b -- n )  + * ; ok
```

РЕШЕНИЕ ЗАДАЧ (УПРАЖНЕНИЕ 2-Б)

Преобразуйте следующие выражения из инфиксной формы в форму определений Форта и укажите порядок аргументов в стеке для этих определений. Порядок аргументов в стеке может быть произвольным, но он должен быть наиболее удобным для данного определения. В соответствии с номером упражнения 2-Б вы можете именовать ваши определения как 2Б1 и 2Б2 и т. д. Например:

1. ab + c примет вид : 2Б1 * + ;

2. $\frac{a - 4b}{6} + c$;

3. $\frac{a}{8b}$;

4. $\frac{0,5ab}{100}$;

5. a (2a + 3);

6. $\frac{a - b}{c}$.

Ответы к упражнению 2-Б

```
2.  : 2Б2  ( c a b -- x )
       4 *  -  6 /  + ;

3.  : 2Б3  ( a b -- x )
       8 *  / ;

4.  : 2Б4  ( a b -- x )
       *  200 / ;

5.  : 2Б5  ( a a -- x )
       2 *  3 +  * ;
```

6. Если вы скажете, что такое выражение преобразовать нельзя, то будете правы, по крайней мере сейчас, пока мы еще не рассмотрели специальных стековых операций.

ОПЕРАЦИИ ДЕЛЕНИЯ

Слово / (слэш) отображает самую простую операцию деления в Форте. *Слэш* обеспечивает только частное; если в результате деления образуется остаток, он теряется. Набрав на клавиатуре

```
22 4 / . 5 ok
```

вы получите только частное (пять) и не получите остаток (два). Если же вы хотите выполнить операцию деления, аналогичную предусмотренной в карманном калькуляторе, такой результат вас не удовлетворит, тем более что Форт не позволяет даже округлить полученное целое число.

Но / — всего лишь одна из нескольких операций деления в Форте. Такое разнообразие операций дает пользователю широкие возможности. Во многих задачах округленный результат или результат двойной точности вам и не требуется. Допустим, вам нужно решить такую задачу: сколько банкнот достоинством в один доллар получается при размене 22 четвертей доллара? Ответ очевиден: пять (а не 5.5). Машинный меняла, к примеру, не будет знать, как выдать вам 5.5 дол. Здесь необходимы операции, подобные операции /, но вычленяющие целые частное и остаток:

```
MOD      ( n1 n2 -- n-остаток )     Деление. В стек помещается
                                    остаток от деления.

/MOD     ( n1 n2 --                 Деление. В стек помещаются
           n-остаток n-частное )    остаток и частное.
```

Итак, если вам требуется только частное, применяйте операцию /, если вам требуется только остаток — операцию **MOD**[1], а если и остаток, и частное — операцию **/MOD**.

Выполним в качестве примера операцию **/MOD**:

```
22 4 /MOD . . 5 2 ok
```



[1] *Для любознательных.* MOD — сокращение от modulo, что означает «остаток».

Полученных вами знаний уже достаточно для того, чтобы легко написать следующий набор определений:

```
: ДОЛ-ЧЕТВЕРТИ  ( четверти -- четверти доллары )   4  /MOD ;
: .ДОЛЛАРЫ  ( доллары -- )   .  ." долларов " ;
: .ЧЕТВЕРТИ  ( четверти -- )   .  ." четверти " ;
: ЧЕТВЕРТИ  ( четверти -- )
     ДОЛ-ЧЕТВЕРТИ  ." Получается"  .ДОЛЛАРЫ  ." и "  .ЧЕТВЕРТИ ;
```

Далее вы можете ввести

```
22 ЧЕТВЕРТИ
```

и получить результат:

```
22 ЧЕТВЕРТИ Получается 5 долларов и 2 четверти ок
```

В Стандарте Форт-83 во всех операциях деления частное *не округляется*[1].

Что же делать, если в прикладных программах требуется округление? Не беспокойтесь — нужные средства легко создаются, как вы увидите в разд. «Округление» гл. 5, путем комбинирования и расширения элементарных арифметических операций.

МАНИПУЛЯЦИИ СО СТЕКОМ

Если вы пытались решить задачу 6 в последнем упражнении, то вам уже ясно, что выражение в инфиксной форме

$$\frac{a - b}{c}$$

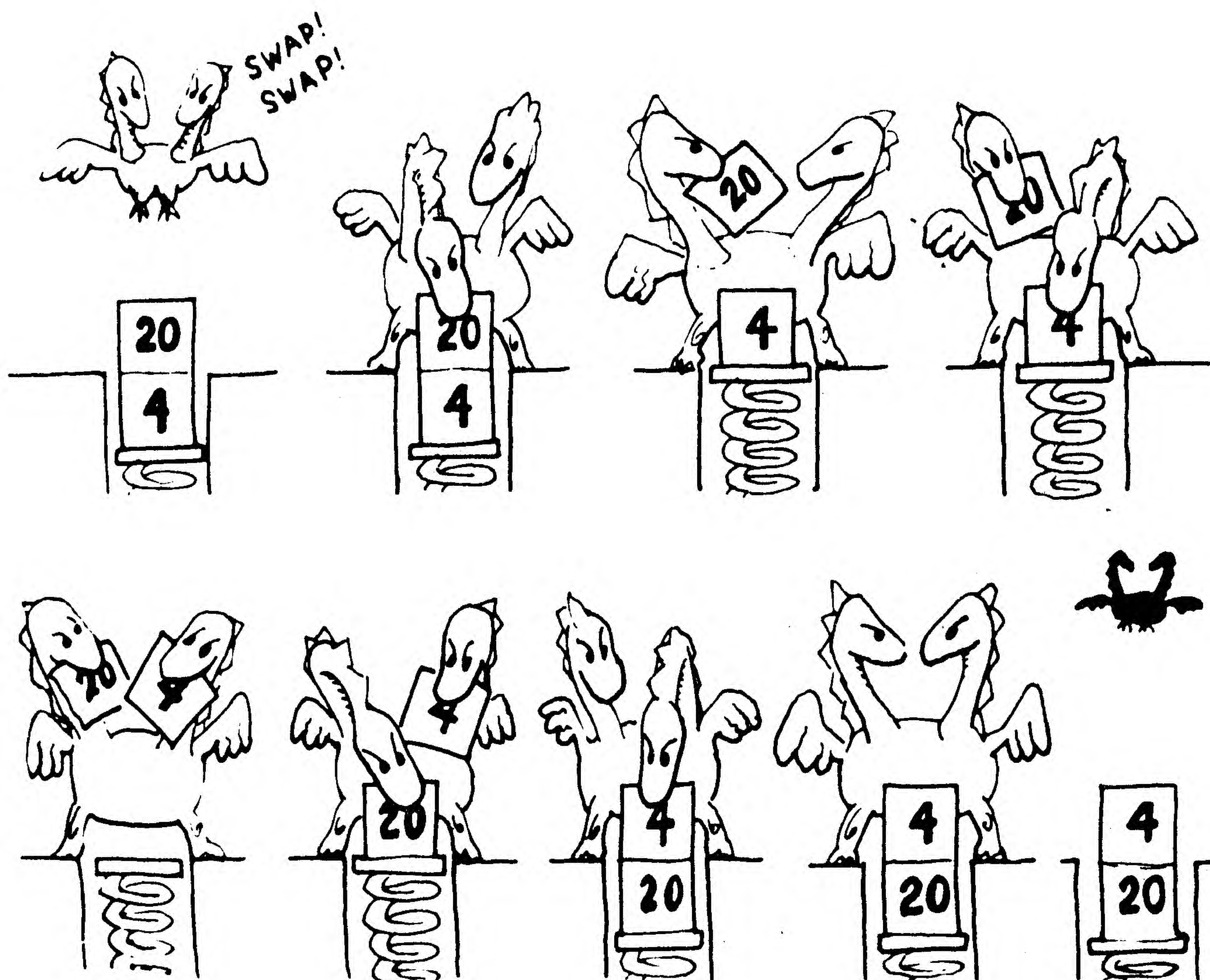
нельзя преобразовать в форму определения Форта без того, чтобы каким-то образом не поменять местами значения в стеке. «Каким-то образом» — это значит, что вы должны выполнить операцию преобразования стека, а именно *перестановку* **SWAP**.

SWAP (ПЕРЕСТАНОВКА)

---

[1] *Для тех, кто имеет склонность к математике.* Хотите верьте, хотите нет, но в информатике мы сталкиваемся с противоречиями при решении даже такой простой задачи, как «—32 разделить на 7». Результатом может быть либо —4 при остатке —3 (—4 × 7 = —28; —28 + —3 = 31), *либо* —5 при остатке 4 (—5 × 7 = —35; —35 + 4 = 31).

Группа по разработке стандарта Форт-83 приняла решение о том, что при выполнении операций деления частное не должно *округляться*. Иными словами, из двух целых чисел, между которыми находится дробное частное, выбирается меньшее. В нашем примере —5 меньше, чем —4, поэтому выбирается —5. При делении без округления частного знак остатка совпадает со знаком делителя. Таким образом, если мы делим —31 на 7 в среде Форт-83, то получаем частное —5 и остаток 4. Это правило относится к делению чисел со знаком и не приводит к противоречиям в окрестности нуля.



Слово **SWAP**, определено так, что при его выполнении два верхних элемента стека меняются местами.

Вы можете проверить, как выполняется операция **SWAP**, а также поэкспериментировать со стеком за своим терминалом в режиме калькулятора, когда это слово не должно появляться внутри определения.

Для начала введите следующее:

```
1 2 . . 2 1 ok
```

а затем то же самое, но со словом **SWAP**:

```
1 2 SWAP . . 1 2 ok
```

Теперь задача 6 из упр. 2-Б может быть решена таким образом:

```
- SWAP /
```

если содержимое стека определяется как (c a b —).

Присвоим переменным a, b, c контрольные значения: a = 10, b = 4, c = 2. Поместим их в стек и выполним предложение, например такое:

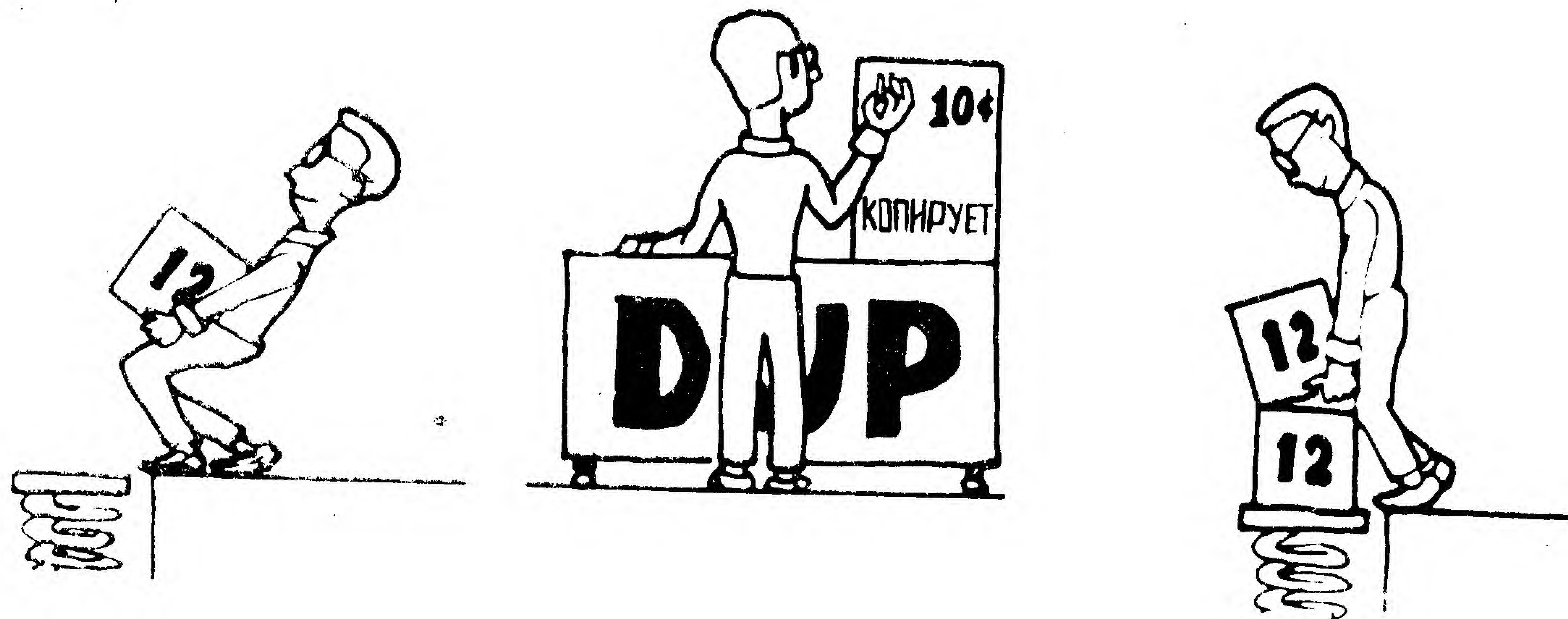
```
2 10 4 - SWAP / . 3 ok
```

Ниже приводится список операций работы со стеком:

SWAP	(n1 n2 -- n2 n1)	Перестановка двух верхних элементов стека.
DUP	(n -- n n)	Дублирование верхнего элемента стека.
OVER	(n1 n2 -- n1 n2 n1)	Копирование второго элемента стека и размещение копии в вершине стека.
ROT	(n1 n2 n3 -- n2 n3 n1)	Размещение третьего элемента в вершине стека.
DROP	(n --)	Удаление верхнего элемента из стека.

DUP (дублирование)

При выполнении следующей в списке операции над стеком, **DUP**, просто создается второй экземпляр верхнего элемента стека. Например, если у вас в стеке есть элемент a, то вы можете вычислить a^2:




```
DUP *
```

При этом выполняются следующие действия:

ОПЕРАЦИЯ	СОДЕРЖИМОЕ СТЕКА
	a
DUP	a a
*	a^2

OVER (ЧЕРЕЗ)

Теперь допустим, что кто-то попросил вас вычислить выражение

```
a * (a + b)
```

при следующем содержимом стека:

```
( a b -- )
```

Вы скажете, что для этого потребуется новая операция со стеком, так как вам нужно два экземпляра a, и a должно находиться под b. **OVER** и есть та самая «новая» операция. **OVER** создает еще один экземпляр a, который «перепрыгивает», как при игре в чехарду, *через* b:

```
( a b -- a b a)
```

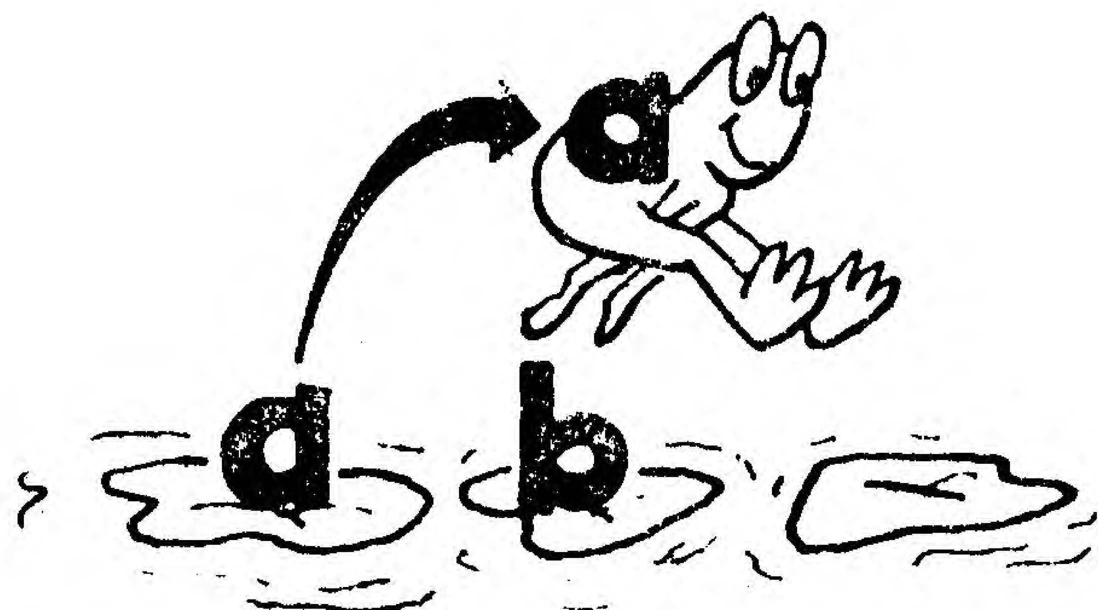
Теперь исходное выражение

```
a * (a + b)
```

может быть легко записано в виде

```
OVER + *
```

При этом происходит следующее:



ОПЕРАЦИЯ	СОДЕРЖИМОЕ СТЕКА
	a b
OVER	a b a
+	a (b+a)
*	a*(b+a)

Прежде чем записывать выражения на Форте, их нужно разложить на множители. Например, вычислить выражение

$a^2 + ab$

с помощью Форта, применяя только описанные выше средства, довольно сложно (если вообще возможно) до тех пор, пока вы не разложите это выражение на множители, т. е. не приведете его к виду

```
a * (a + b)
```

А такое выражение вы уже умеете вычислять

ROT (ПЕРЕМЕЩАТЬ ПО КРУГУ)

Это четвертая операция в нашем списке (**ROT** — сокращение от **rotate**). Посмотрите, что происходит при ее выполнении с тремя верхними элементами стека:



Например, если вам нужно вычислить выражение
ab — bc,
то сначала необходимо вынести b за скобки:

```
b * (a - c)
```

а затем, если начальное состояние стека таково:

```
( c b a -- )
```

можно написать

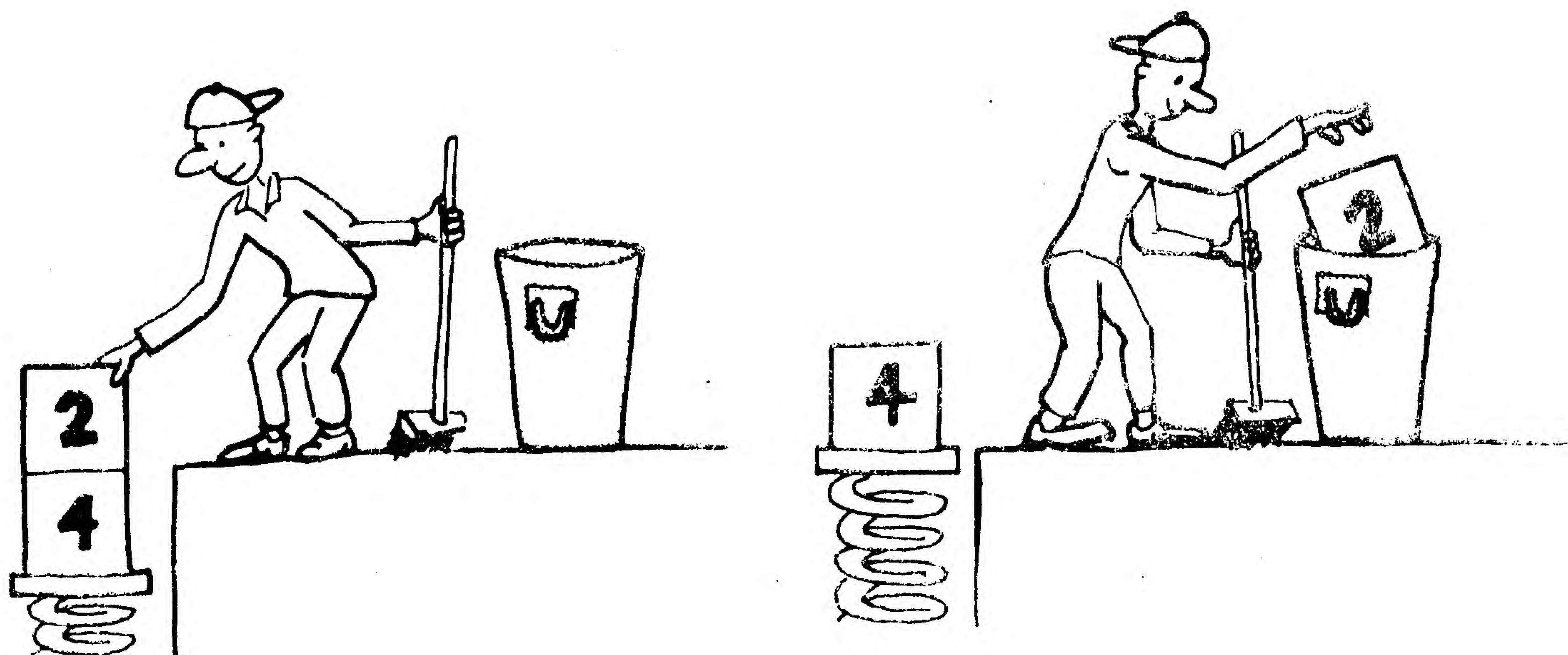
```
ROT - *
```

При этом выполняются следующие действия:

ОПЕРАЦИЯ	СОДЕРЖИМОЕ СТЕКА
	c b a
ROT	b a c
-	b (a - c)
*	(b*(a-c))

DROP (ИСКЛЮЧИТЬ/УДАЛИТЬ)

Последняя в нашем списке операция со стеком — **DROP**. Ее назначение состоит в том, чтобы удалить верхний элемент из стека.



Просто, не правда ли? Позднее мы найдем для этой операции несколько хороших применений.

ПЕЧАТЬ БЕЗ ИЗМЕНЕНИЯ СОДЕРЖИМОГО СТЕКА

При отладке программ программисту часто необходимо знать, что в данный момент находится в стеке. Конечно, набрав серию точек, мы выведем на терминал содержимое стека, но при этом выведенные значения будут потеряны для последующих операций. В большинстве Форт-систем имеется слово с именем .S, которое для своего выполнения не требует аргументов в стеке и выводит содержимое стека, не разрушая его, т. е. оставляя после завершения функционирования прежнее состояние стека. При отладке программы вы можете ввести несколько слов, выполнить слово .S, чтобы убедиться в том, что содержимое стека соответствует результату функционирования введенных слов, ввести еще несколько слов, снова выполнить .S и т. д.

Давайте проверим:

```
1 2 3 .S
1 2 3 ok

ROT .S
2 3 1 ok
```

При изучении слов манипулирования со стеком начинающие иногда используют следующий прием:

```
: SWAP   SWAP .S ;
: DUP   DUP .S ;
: OVER   OVER .S ;
: ROT   ROT .S ;
: DROP   DROP .S ;
```

Таким образом осуществляется немедленная обратная связь по каждой выполняемой команде. (Это возможно потому, что интерпретатор из двух слов с одинаковыми именами выполняет занесенное в словарь последним.)

ЗАДАЧИ НА ВЫПОЛНЕНИЕ ОПЕРАЦИЙ СО СТЕКОМ И АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ (УПРАЖНЕНИЕ 2-В)

1. Напишите предложение, которое позволит расположить три элемента стека в обратном порядке: вместо a b c — c b a.

2. Напишите предложение, аналогичное по результату выполнения операции **OVER** (но без использования **OVER**).

Напишите определения для следующих выражений, что должно привести к указанному состоянию стека (по его исходному состоянию):

3. $\frac{n+1}{n}$ (n — результат).

4. $x(7x+5)$ (x — результат).

5. $9a^2 - ba$ (a b — результат).

Ответы к упражнению 2-В

```
1.    SWAP ROT

2.    SWAP DUP ROT SWAP

3.    : 2B3    DUP 1 +  SWAP / ;     или
      : 2B3    DUP 1+  SWAP / ;

4.    : 2B4    DUP 7 *  5 + * ;

5.    : 2B5    OVER  9 *  SWAP -  * ;
```

ДВОЙНЫЕ ОПЕРАЦИИ[1]

Следующие четыре операции должны вам показаться знакомыми:

2SWAP	(d1 d2 -- d2 d1)	Перестановка двух верхних пар элементов стека.
2DUP	(d -- d d)	Дублирование верхней пары элементов стека.
2OVER	(d1 d2 -- d1 d2 d1)	Копирование второй пары элементов стека и размещение копии в вершине стека.
2DROP	(d --)	Удаление верхней пары элементов из стека.

Префикс 2 означает, что перечисленные стековые операции выполняются над парами чисел[2], а буква d, используемая в стековой нотации, — что вместо нее подставляется «двойное» число.

---

[1] Стандарт Форт-83. Эти слова входят в «Дополнительный перечень слов двойной точности».

[2] *Для специалистов.* Эти операции также могут выполняться над числами двойной длины (32 бита, или разряда).

Это понятие имеет вполне конкретный смысл, который объясняется в гл. 7.

Операции над двойными числами настолько очёвидны, что нет необходимости приводить примеры на их выполнение. Заметим лишь, что кроме перечисленных существуют еще несколько операций, о которых здесь еще не упоминалось, поэтому не пытайтесь самостоятельно работать со стеком, так как вы будете выполнять много ненужных действий, в чем и убедитесь впоследствии.

Ниже приводится перечень слов Форта, которые были введены в данной главе:

```
+       ( n1 n2 -- сумма)          Сложение.

-       ( n1 n2 -- разность)       Вычитание (n1-n2).

*       ( n1 n2 -- произвед)       Умножение.

/       ( n1 n2 -- частное)        Деление (n1/n2).

MOD     ( n1 n2 -- n-остаток)      Деление. В стек заносится остаток
                                   от деления.

/MOD    ( u1 u2 --                 Деление. В стек заносятся оста-
          n-остаток n-частное)     ток и частное.

SWAP    ( n1 n2 -- n2 n1)          Перестановка двух верхних  элементов
                                   стека.

DUP     ( n -- n n)                Дублирование верхнего элемента стека.

OVER    ( n1 n2 -- n1 n2 n1)       Копирование второго элемента и разме-
                                   щение копии в вершине стека.

ROT     ( n1 n2 n3 --              Размещение третьего элемента в верши-
               n2 n3 n1)           не стека.

DROP    ( n -- )                   Удаление из стека верхнего элемента.

2SWAP   ( d1 d2 -- d2 d1)          Перестановка двух верхних пар чисел.

2DUP    ( d -- d d)                Дублирование пары чисел, находящейся
                                   в вершине стека.

2OVER   ( d1 d2 -- d1 d2 d1)       Копирование второй пары чисел и разме-
                                   щение копии в вершине стека.

2DROP   ( d -- )                   Удаление из стека верхней пары элемен-
                                   тов.
```

ОСНОВНЫЕ ТЕРМИНЫ

Числа двойной длины. Целые числа в диапазоне от —2 биллионов до +2 биллионов (будут строго введены в гл. 7).

Числа ординарной длины. Целые числа в диапазоне от —32 768 до +32 767. Только эти числа можно использовать в качестве аргументов и получать в виде результата при выполнении любой из

рассмотренных выше операций. (Этот на первый взгляд выбранный произвольно диапазон определяется структурой разработки компьютеров, как вы увидите позднее.)

УПРАЖНЕНИЯ

2.1. В чем различие между **DUP DUP** и **2DUP**?

2.2. Определите слово NIP (отщипнуть) для удаления второго элемента стека, т. е.

```
( a b -- b)
```

2.3. Определите слово TUCK (подобрать) для копирования, верхнего элемента стека и размещения копии в стеке третьим элементом, т. е.

```
( a b -- b a b)
```

2.4. Определите слово —ROT, которое размещало бы верхний элемент под вторым и третьим (в противоположность **ROT**), т. е.

```
( a b c -- c a b)
```

2.5. Напишите предложение для перестановки четырех верхних элементов стека в обратном порядке, т. е.

```
( 1 2 3 4 -- 4 3 2 1)
```

2.6. Напишите слово с именем 3DUP, которое будет дублировать три верхних элемента стека, например

```
( 1 2 3 -- 1 2 3 1 2 3)
```

Напишите определения для следующих выражений в инфиксной форме с учетом указанной стековой нотации:

2.7. $a^2 + ab + c$ (c a b — результат).

2.8. $\frac{a - b}{a + b}$ (a b — результат).

2.9. Представьте себе, что вы программист, занимающийся учетом продукции на птицеферме Мерайи. Определите слово с именем УПАКОВКА, которое снимает со стека значение, равное числу яиц, снесенных в день подсчета на данной ферме. В результате его выполнения на печать выдается число коробок, требуемых для упаковки этих яиц, из расчета по 12 штук на коробку, а также число яиц, оставшихся неупакованными из-за того, что их недостаточно для заполнения еще одной коробки.

3 Заказ № 676

Глава 3
КАК РАБОТАТЬ НА ФОРТЕ

До сих пор вы компилировали новые определения в словарь, набирая их на клавиатуре вашего терминала. Здесь же вам предлагается другой вариант создания определений — с использованием дисковой памяти и текстового редактора Форта.

Настоящая глава состоит из двух частей. В первой части обсуждаются вопросы применения дисковой памяти для хранения исходных текстов программ на Форте, а также правила оформления этих исходных текстов. Первая часть рассчитана на читателей с любым уровнем подготовки. Во второй части рассматривается конкретный текстовый редактор. Если текстовый редактор, с которым вы работаете, отличается от рассматриваемого, мы рекомендуем вам обратиться к документации по вашей Форт-системе.

Часть 1
ОБЩИЕ СВЕДЕНИЯ

ЕЩЕ РАЗ О СЛОВАРЕ

Поработав с настоящим компьютером, вы, возможно, сделали бы для себя кое-какие открытия, о которых еще не упоминалось.

Открытие первое: вы можете дать одному и тому же слову несколько определений, приписывая ему всякий раз новый смысл, но выполняться будет только последнее определение.

Например, если вы определили слово ВСТРЕЧА:

```
: ВСТРЕЧА   ." Привет.  Я говорю на Форте. " ; ok
```

то при его выполнении получите следующий результат:

```
ВСТРЕЧА привет.  Я говорю на Форте. ok
```

Но если вы переопределите это слово:

```
: ВСТРЕЧА   ." Алло, я слушаю вас! " ok
```

то при его выполнении сработает более позднее определение:

```
ВСТРЕЧА Алло, я слушаю вас! ок
```

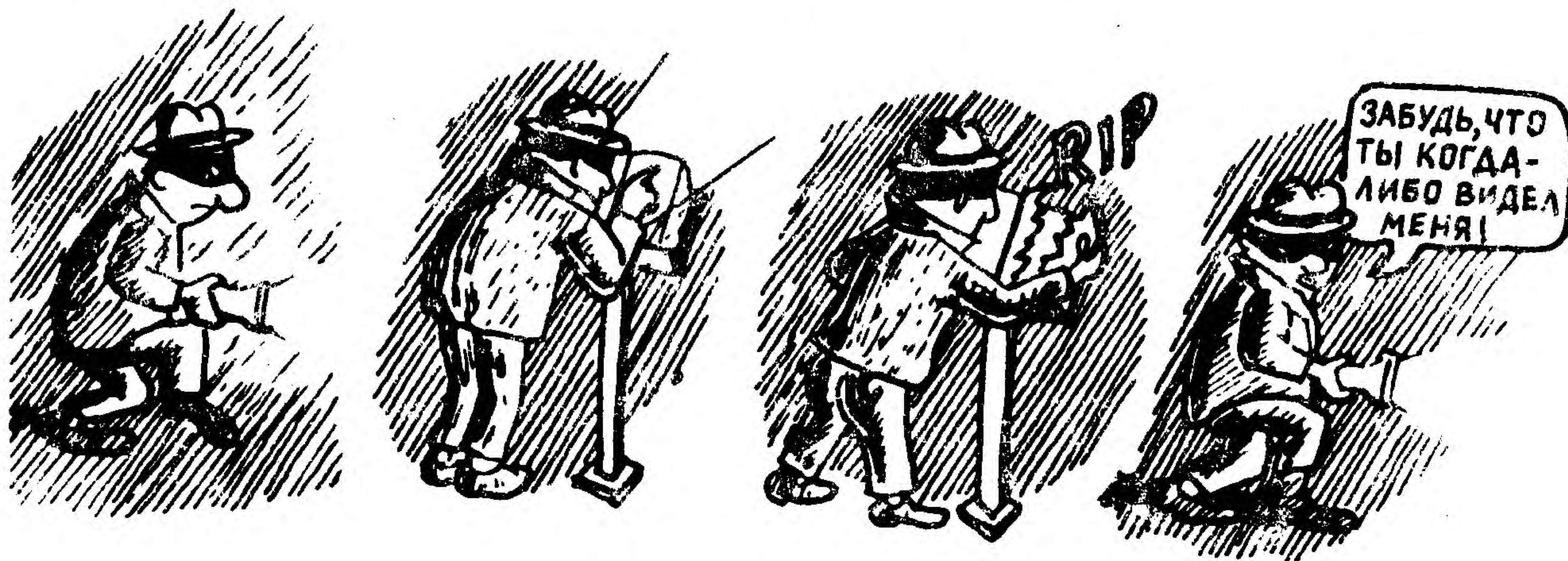
Исчезло ли первое определение ВСТРЕЧА? Нет, оно сохранилось, но текстовый интерпретатор всегда начинает просмотр словаря с конца — с элемента, занесенного в словарь последним. Из нескольких определений с одним и тем же именем интерпретатор передает слову **EXECUTE** первое встретившееся. Вы можете убедиться в том, что прежнее определение ВСТРЕЧА все еще находится в словаре. Для этого наберите на клавиатуре

```
FORGET ВСТРЕЧА ок
```

и

```
ВСТРЕЧА Привет.  Я говорю на Форте. ок
```

(Действует снова старое определение!)



Слово **FORGET** (ЗАБЫТЬ) ищет указанное слово в словаре и удаляет из словаря (это его основная функция) само слово, а также все то, что вы успели определить после него. **FORGET**, как и интерпретатор, начинает свой поиск с конца: он удаляет только последний вариант определения данного слова (вместе со всеми словами, специфицированными после него). Поэтому теперь, если вы наберете на клавиатуре слово ВСТРЕЧА, интерпретатор найдет первоначальное слово ВСТРЕЧА. **FORGET** — очень полезное слово. Оно помогает вам очищать ваш словарь во избежание его переполнения. (Словарь занимает память, а мы должны ее экономить.)

Открытие второе: если вы вводите определение с помощью клавиатуры (как вы сейчас это делаете), то исходный текст[1] его не сохраняется.

---

[1] *Для начинающих.* Исходным текстом называется текстовый вариант определения, например:

```
: ПЛЮС-ЧЕТЫРЕ   ( n -- n+4)     4 + ;
```

Этот первоначальный, исходный, вариант преобразуется в словарную форму и становится элементом словаря.

В словаре запоминается только скомпилированная форма вашего определения. А как быть, если вы захотите внести изменение в уже определенное слово? Вы должны повторно набрать полностью все определение, внеся в него соответствующие изменения. Это может показаться вам достаточно утомительным, не так ли? И даже хуже:

Открытие третье: если вы выключите компьютер, а затем снова включите его, то все набранные вами определения исчезнут.

Очевидно, что необходим какой-то способ запоминания исходного текста с тем, чтобы его можно было изменять и перекомпилировать в любое время. Здесь-то и приходит вам на помощь *редактор*. Он позволяет запомнить ваш исходный текст на диске. Поэтому давайте выясним, что представляет собой диск и как Форт-система работает с ним.

ИСПОЛЬЗОВАНИЕ ДИСКОВОЙ ПАМЯТИ

Почти все Форт-системы используют дисковую память. Чтобы понять, что такое дисковая память, ее можно сравнить с оперативной памятью (ЗУПВ — запоминающее устройство с произвольной выборкой). Они различаются между собой примерно так же, как стационарная картотека (шкаф) и вращающаяся. До сих пор вы имели дело с памятью компьютера, которая подобна вращающейся картотеке. Компьютер может получить доступ к этой памяти почти мгновенно, поэтому программы, хранимые в ОЗУ, выполняются очень быстро. К сожалению, объем такой памяти ограничен и ее использование обходится относительно дорого. Кроме того, при выключении компьютера содержимое памяти теряется.



С другой стороны, диск называют устройством с большим объемом памяти, так как аналогично шкафу с картотекой он может хранить значительный объем информации при меньших затратах на единицу данных, чем оперативная память компьютера. Как и

магнитная лента, диск сохраняет информацию при отключении питания.

Когда вы определяете некоторое слово, компилятор помещает определение этого слова в ЗУПВ, чтобы доступ к нему осуществлялся достаточно быстро. Постоянным же местом хранения исходного текста данного определения является диск. Вы можете средствами Форта записать ваш исходный текст на диске, а впоследствии считывать его с диска и передавать для обработки текстовому интерпретатору[1].

Дисковая память Форт-системы подразделяется на так называемые *блоки*. Храниться на диске в виде блоков может любая информация. Существует соглашение, по которому блоки, содержащие исходный текст, вмещают 1024 символа и состоят из 16 строк по 64 символа каждая, что согласуется с размерами экрана вашего компьютера. (В некоторых случаях для обозначения блока с исходным текстом применяется термин *экран*. Для простоты мы будем пользоваться только термином *блок*.)

Блок с исходным текстом выглядит следующим образом:

```
Block  50
   0 ( Большая буква "F" )
   1 : STAR  42 EMIT ;
   2 : STARS  ( Количество )   0 DO  STAR  LOOP ;
   3 : MARGIN   CR  30 SPACES ;
   4 : BLIP   MARGIN STAR ;
   5 : BAR   MARGIN  5 STARS ;
   6 : F   BAR BLIP BAR BLIP BLIP  CR ;
   7
   8
   9 F
  10
  11
  12
  13
  14
  15
```

Верхняя строка (Block 50) и числа слева от текста на диске отсутствуют. Они выводятся словом **LIST**, чтобы вам было легче ориентироваться. Для того чтобы вывести требуемый блок, наберите номер этого блока и слово **LIST** (РАСПЕЧАТАТЬ), например:

```
50 LIST
```

---

[1] *Для начинающих.* Диск иногда используют для хранения самой Форт-системы. Когда вы активируете, или *загружаете*, Форт, это, как правило, означает его копирование из некоторого участка дисковой памяти в ЗУПВ, где он представляется в виде словаря. (В других системах Форт хранится в постоянной памяти, предназначенной только для чтения из нее (ПЗУ). Здесь он становится доступным сразу после включения питания.)

Как ввести в блок исходный текст? Для этого существует редактор. Поскольку каждая Форт-система снабжена собственным редактором, за разъяснениями обращайтесь к документации по вашей системе. (Один из таких редакторов описан во второй части настоящей главы.)

Для простоты предположим, что блок 50 содержит определения, рассмотренные выше. Почти все они вам знакомы: это определения, которые вы использовали для вывода на дисплей большой буквы F. Теперь, имея данные определения в блоке, как передать блок 50 текстовому интерпретатору? С помощью выражения

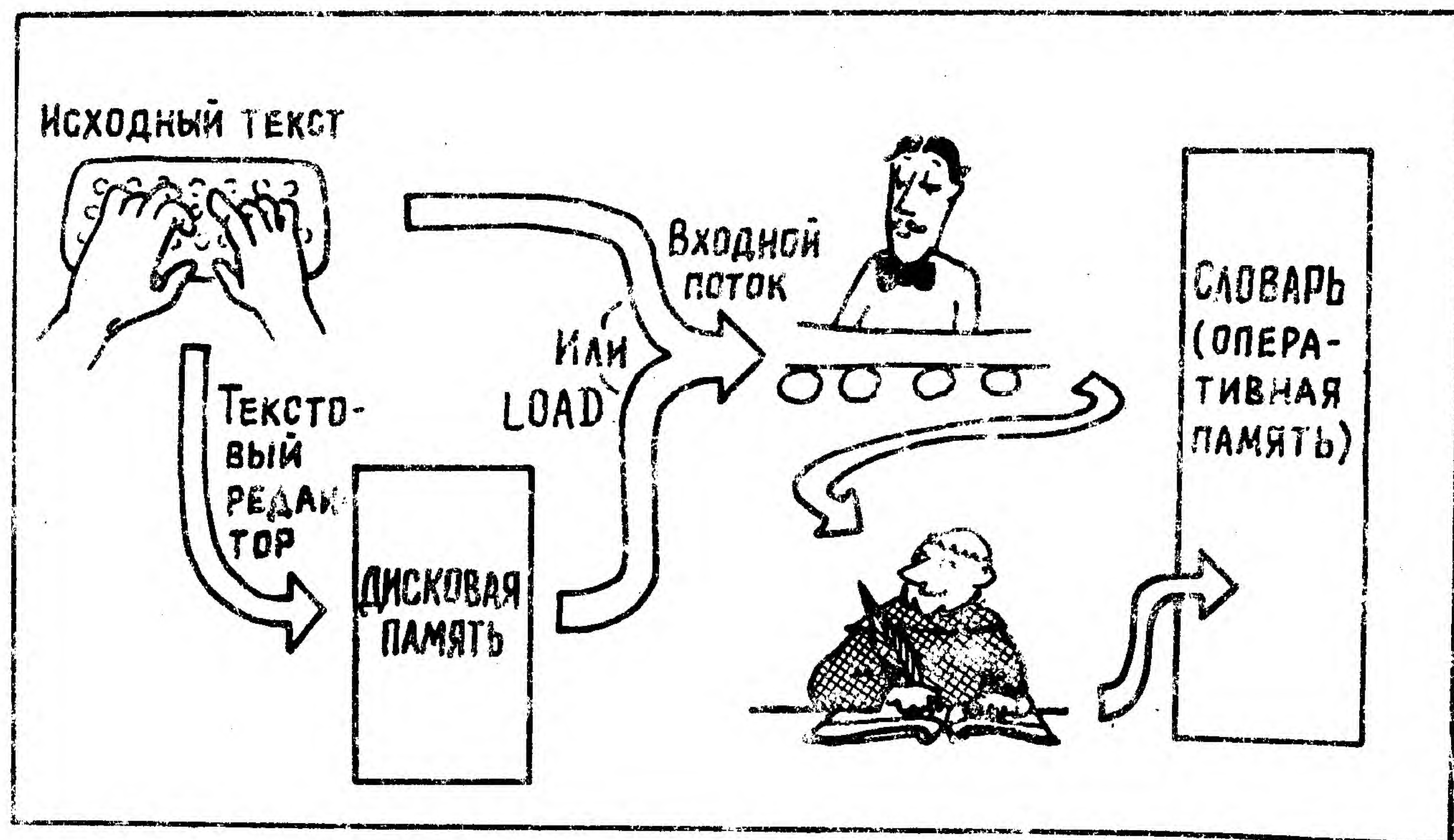
```
50 LOAD
```

Слово **LOAD** (ЗАГРУЗИТЬ) передает указанный блок текстовому интерпретатору, что приводит к компиляции всех находящихся в блоке определений.

После загрузки определений вы можете набрать новое слово F и получить ожидаемый результат. Но заметьте, что мы уже поместили наше новое слово в строку 9 для того, чтобы показать, что при загрузке блока выполняется его содержимое. При загрузке именно *этого* блока не только производится компиляция определений, составляющих слово F, но и само слово F будет выполнено и вы увидите букву F на вашем экране.

Как уже отмечалось, текстовый интерпретатор сканирует *входной поток*. Ранее мы считали, что входным потоком является строка, вводимая с клавиатуры. Теперь вам известно, что можно *переключать* источник входного потока. Как правило, входной поток поступает с клавиатуры дисплея; когда же вы применяете слово **LOAD**, входной поток поступает с диска.

Итак, у входного потока два источника: непосредственно клавиатура или диск.



В гл. 10 мы более подробно обсудим использование Форт-системой дисковой памяти. Но прежде чем вы приступите к редактированию исходного текста, введем еще одну команду. Слово **FLUSH** (ВЫБРОС) гарантирует, что все изменения, которые вы пытались внести в некоторый блок, действительно были записаны на диск.

Когда вы редактируете конкретный блок, любое вносимое вами исправление *не* поступает на диск немедленно. На самом деле вы работаете с некоторой копией этого блока, расположенного в каком-то участке ЗУПВ. В конечном итоге по завершении ваших исправлений Форт-система возвратит откорректированную копию блока на диск. Но представьте себе, что вы отключили компьютер до того, как тот вернул копию блока на диск. Или, предположим, вы сменили диски. Кроме того, в результате допущенной вами программной ошибки вы могли до переноса Фортом ваших исправлений на диск просто испортить систему.

По крайней мере до тех пор, пока вы не прочитаете гл. 10, прежде чем снять диск, отключить питание или предпринять что-нибудь опасное, обязательно введите слово **FLUSH**[1].

Ниже приводится перечень команд, введенных в данном разделе.

FORGET	имя	(—)	С помощью этого слова мы забываем (удаляем из словаря) указанное слово и все слова, внесенные в словарь после него.
LIST		(n —)	Вывод на экран дискового блока.
LOAD		(n —)	Загрузка дискового блока (компиляция или выполнение). Блок 0 обычно загружен быть не может.
FLUSH		(—)	Запись всех обновленных дисковых буферов на диск, после чего освобождение этих буферов.

ПРАВИЛА ЗАПИСИ ФОРТ-ПРОГРАММ

Вернемся к нашему учебному блоку из предыдущего раздела. Хороший стиль программирования на Форте требует отводить строку 0 любого блока под *комментарий* — краткое описание функционального назначения определений, содержащихся в данном блоке. В комментарий часто включают дату внесения последнего изменения и инициалы программиста.

---

[1] *Относительно Форта, функционирующего под управлением других операционных систем.* Многие Форт-системы работают под управлением других операционных систем, например, CP/M или MS-DOS. В них блоки Форта представляют собой участки размером в 1К в специально зарезервированном для этих целей файле (или файлах). Как открывать и закрывать такие файлы, объясняется в документации по вашей системе.

Мы использовали в приведенном выше примере для выделения комментария круглые скобки. В некоторых системах имеется слово **/** (пропуск строки), которое предписывает интерпретатору пропустить остаток строки (все что находится справа от этого слова). По аналогии со словом **(** слово **/** начинает комментарий, но в отличие от него не требует ограничителя. Вы можете помещать **/** в середину любой строки: ее содержимое слева от этого слова будет обработано интерпретатором, справа — нет.

Похожее слово **/S** (пропуск экрана) предписывает интерпретатору игнорировать дальнейший текст в данном блоке. В отдельных Форт-системах для таких целей применяется слово **EXIT**.

Ниже приводятся еще несколько правил, позволяющих сделать текст в блоке удобочитаемым.

1. Отделяйте комментарий с обеих сторон двумя пробелами. Если стековый комментарий отсутствует, отделяйте имя определения от содержательной части тремя пробелами.

2. Разбивайте определения на фрагменты, разделяемые двумя пробелами.

3. Если определение занимает более одной строки, делайте отступ во всех строках, кроме первой.

4. Не помещайте на одной строке более одного определения.

5. Определения должны быть краткими! В среднем определение должно занимать две строки.

В книге автора «Думаем на Форте» [1] хорошему стилю программирования на Форте посвящена целая глава.

Итак, запомните две команды, введенные в этом разделе:

```
\     ( -- )          Пропуск оставшегося текста данной строки.

\S    ( -- )          Пропуск оставшегося текста экрана.
```

ОСОБЕННОСТИ ПРОГРАММИРОВАНИЯ НА ФОРТЕ

Программист пишет программу на Форте обычно в несколько этапов:

1. С помощью редактора Форта он делает доступным некоторый блок.
2. Набирает определение.
3. Завершает работу с редактором (при необходимости).
4. Загружает данный блок.
5. Проверяет новое слово.
6. Если слово выполняется неправильно, то программист забывает это определение посредством **FORGET**, редактирует его и повторно загружает. Если же слово выполняется правильно, программист возвращается к п. 1 и принимается за следующее определение.

Основной причиной, по которой программирование на Форте происходит быстрее, чем на других языках, является быстрая оборачиваемость цикла «кодирование—загрузка—тестирование». Загрузка блока занимает менее секунды.

Поэтому программирование на Форте требует несколько иной стратегии, чем на большинстве известных языков программирования, например Си. После того как вы представили себе программу в целом и решили, что будет выполнять каждое слово в отдельности, можете приступать к кодированию. У вас есть возможность проверять ваши определения по мере написания и модифицировать их с целью удаления ошибок, дальнейшего совершенствования и наведения «глянца». Таким образом, все последующие слова создаются на проверенном фундаменте и вы управляете программой с самого начала ее создания. (В книге автора «Думаем на Форте» такой подход называется итеративной разработкой.)

ЗАГРУЗКА ПРОГРАММ

Как вы уже знаете, процесс кодирования может потребовать неоднократной повторной загрузки одного и того же блока или блоков. Но учтите: всякий раз, производя загрузку определений, вы увеличиваете размеры вашего словаря.

Допустим, вы загружаете несколько раз некоторый блок, изменяя в нем по одному определению. В результате ваш словарь будет содержать по варианту каждого слова данного блока для каждой загрузки. Простейший способ избежать этого состоит в применении слова **FORGET**. Например, если вы только что внесли исправления в определение слова F в блоке 50 и хотите вновь загрузить последний, то вы должны набрать на клавиатуре следующее:




```
FORGET STAR ok
50 LOAD ok
```

Помните, что слово **FORGET** забывает само указанное слово и все, что было определено после него.

Возможна ситуация, когда из нескольких существующих вариантов одной и той же программы в конкретный момент времени требуется компилировать только один из них. Предположим, вы должны написать программу обработки слов. После создания основы программы вы хотите присоединить к ней фрагменты, по-

зволяющие использовать эту программу в одном случае для обработки корреспонденции, в другом — для работы с журнальными статьями, а в третьем — для обработки адресных наклеек.



В Форте три перечисленных варианта фрагментов называются *оверлейными структурами*[1], так как они при исполнении не пересекаются и взаимозаменяемы в памяти. Рассмотрим их более подробно.

Последнее определение основы программы должно содержать только имя, например:

```
: VARIATIONS ;
```

Это определение называется *нулевым*, поскольку оно всего лишь отмечает место в вашем словаре. Включите в начало каждого альтернативного фрагмента выражение

```
FORGET VARIATIONS   : VARIATIONS ;
```

С помощью **FORGET** каждый раз при загрузке любого из альтернативных фрагментов система забывает все определения начиная с конца словаря до нашего нулевого определения и вновь компилирует нулевое определение, а затем определения, относящиеся к данному фрагменту. Загружая другой фрагмент, вы тем самым замещаете первый оверлейный сегмент вторым[2].

---

[1] *Для специалистов*. Более точно — *оверлейными структурами компиляции*. Некоторые Форт-системы предлагают возможность загрузки так называемых *двоичных оверлейных структур*, представляющих собой предварительно скомпилированные программные разделы, которые могут быть присоединены к резидентной части словаря.

[2] *Для работающих с системами, в которых есть слово* **EMPTY**. Слово **EMPTY** «забывает» все созданные вами определения. В мультипрограммной системе они составляют лишь ваше собственное расширение словаря.

Мы ввели команду **LOAD** для загрузки одного блока, а как загружать программу, состоящую из нескольких блоков? Используйте слово **THRU**, которое загружает заданный диапазон блоков. Например, выражение

```
180 189 THRU
```

означает загрузку каждого блока, начиная со 180-го и кончая 189-м. Многие системы при выполнении слова **THRU** выводят номера загружаемых блоков, что помогает вам следить за ходом загрузки.

Существует еще один пример. Вам известно, что слово .", помещенное в определение, приводит к выводу некоторого сообщения при выполнении данного определения. Другое слово **.(** можно использовать *за пределами* определения. Наберите на клавиатуре

```
.( Что со мной, доктор?)
```

и нажмите клавишу RETURN. Вы увидите, что набранный текст высветится на экране. Это слово применяется в тех случаях, когда требуется сообщение от некоторого блока во время его загрузки. Например:

```
Block# 10
   0 \ Моя программа
   1
   2 CR  .( Загрузка моей программы...)
   3 20 37 THRU   \ ШТУЧКИ
   4 40 45 THRU   \ ДРЮЧКИ
   5 50 58 THRU   \ ШПУЛЬКИ
   6
   7 CR  .( Моя программа загружена )
   8
   9
```

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАБОТЫ С БЛОКАМИ

Мы знаем, что слово **LIST** инициирует печать только одного блока. Введем несколько слов, имеющихся в большинстве Форт-систем, для вывода группы блоков и для доступа к блокам.

Слово **TRIAD** выводит группу из трех смежных блоков, начиная с блока, номер которого делится на три без остатка. Например, выражение

```
30 TRIAD
```

обеспечит вывод блоков с номерами 30, 31 и 32. (Обращение к **TRIAD** с аргументами 31 и 32 приведет к выдаче той же самой триады блоков.)

Слово **SHOW** выводит группу блоков со смежными номерами. Например, выражение

```
30 38 SHOW
```

приведет к выводу трех триад блоков, начинающихся с номеров 30, 33 и 36 соответственно.

Эти слова применяются для вывода листингов прикладных программ на печатающее устройство. Каким образом осуществляется вывод? К сожалению, все Форт-системы реализуют его по-разному. Перед работой вы должны ознакомиться с документацией по своей системе. Можем предложить в качестве типовых примеров следующие выражения:

· В мультизадачной системе, как правило, вы должны ввести

```
PRINT  17 TRIAD
```

Слово **PRINT** (ПЕЧАТЬ) передает остаток строки, содержащей эту команду, *задаче* вывода на печатающее устройство, а выполнение терминальной задачи продолжается обычным путем.

· В однозадачных системах указанные выше слова переназначают вывод на периферийные *устройства* (не задачи). В подобной системе вы можете ввести такое выражение:

```
PRINTER  17 LIST  CONSOLE
```

В результате вывод назначается на печатающее устройство, блок распечатывается, после чего вывод снова назначается на дисплей. Вы можете промоделировать синтаксис, принятый в мультизадачной системе, следующим образом:

```
: PRINT   PRINTER  INTERPRET  CONSOLE ;
```

В ряде систем слово **SHOW** автоматически назначает вывод на печатающее устройство, а после завершения вывода возвращает назначение на дисплей.

Еще одним важным словом является **INDEX** (КАТАЛОГ), которое распечатывает в блоках из заданного диапазона только строки комментария (нулевые). Например, выражение

```
30 38 INDEX
```

инициирует вывод комментария, содержащегося в блоках с 30-го по 38-й включительно.

Ниже приводится список рассмотренных нами команд.

Если вы еще не знакомы с редактором Форта, то, пожалуй, настало время с ним познакомиться. Возможно, что редактор, имеющийся в вашей системе, отличается от описываемого далее. В этом

THRU	(нач кон --)	Загрузка всех блоков с номерами из диапазона от н а ч до к о н .
.(текст)	(--)	Вывод т е к с т а сообщения, ограниченного правой круглой скобкой. Используется, как правило, за пределами определения через двоеточие.
TRIAD	(n --)	Вывод трех блоков с номерами, включающими n, начиная с номера, делящегося без остатка на 3.
SHOW	(нач кон --)	Вывод блоков с номерами из диапазона от н а ч до к о н по три блока.
INDEX	(нач кон --)	Вывод комментария только для блоков, номера которых входят в диапазон от н а ч до к о н .

случае пропустите оставшуюся часть главы и изучите его по документации к своей системе.

Часть 2

ТЕКСТОВЫЙ РЕДАКТОР ФОРТА

Здесь мы рассмотрим вариант текстового редактора, имеющегося в ряде Форт-систем. Этот вариант ориентирован в большей степени на управление с помощью команд (командный редактор), нежели на управление посредством курсора (экранный редактор). Редактор второго вида проще. Он отличается от первого способом доступа к модифицируемому тексту. Вы «входите» в редактор неким присущим только данной системе способом и вносите исправления в текст, перемещая курсор по экрану нажатием функциональных клавиш (или с помощью устройства типа «мышь», если оно имеется).

В командном редакторе текст редактируемого блока остается неизменным в верхней части экрана. Команды редактирования текста вы вводите в оставшемся пространстве экрана. Для редактирования текста в Форте предусмотрен набор специальных слов. Несколько команд из такого набора позволяют подводить курсор либо к строке с заданным номером, либо, с помощью некоторого шаблона, к строке с заданным текстом. Редактор по конкретной команде находит место в тексте, предназначенное для внесения исправлений, как правило, быстрее программиста, который должен нажимать клавиши манипулирования курсором и следить на экране за положением курсора. Другие команды позволяют вставлять или удалять строки с сохранением образцов для многократных вставок/удалений. При этом лучше всего оставаться «в Форте». Вы можете осуществить перевод из шестнадцатеричной системы в десятичную или же загрузить блок, который только что отредактировали, не выходя из Форт-системы, что осо-

бенно важно для среды, в которой разрабатываются программы. Кроме того, командные редакторы легко расширяемы.

По мнению автора, рассматриваемый ниже редактор удобен, практичен, гибок и ориентирован на Форт в большей степени, чем экранный редактор.

(Следует заметить, что некоторые ранние Форт-системы снабжены так называемым построчным редактором. Вместо того чтобы работать полным экраном, они высвечивают только модифицируемую строку. Поскольку теперь программисты применяют дисплеи, а не телетайпы, построчные редакторы устарели. Тем не менее последние относятся к редакторам командного типа и используют команды, аналогичные описанным в настоящей главе. Поэтому все изложенное здесь в принципе применимо и к ним.)

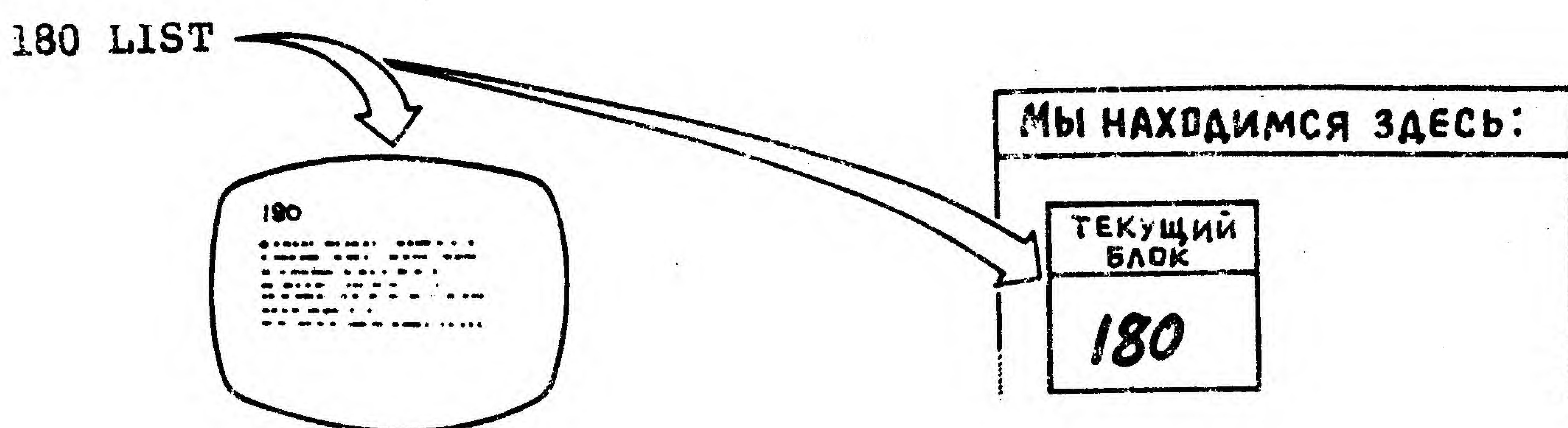
ПРОГРАММА-РЕДАКТОР

Для начала найдите пустой блок (если нужно, используйте **INDEX**) и распечатайте его следующим образом:

```
180 LIST
```

При распечатке пустого блока вы увидите слева на экране 16 строк (0—15), пронумерованных сверху вниз, без какой-либо информации. Приглашение ok в последней строке означает, что текстовый интерпретатор выполнил вашу команду на распечатку данного блока.

Распечатывая блок, вы тем самым выбираете его для дальнейшей работы:



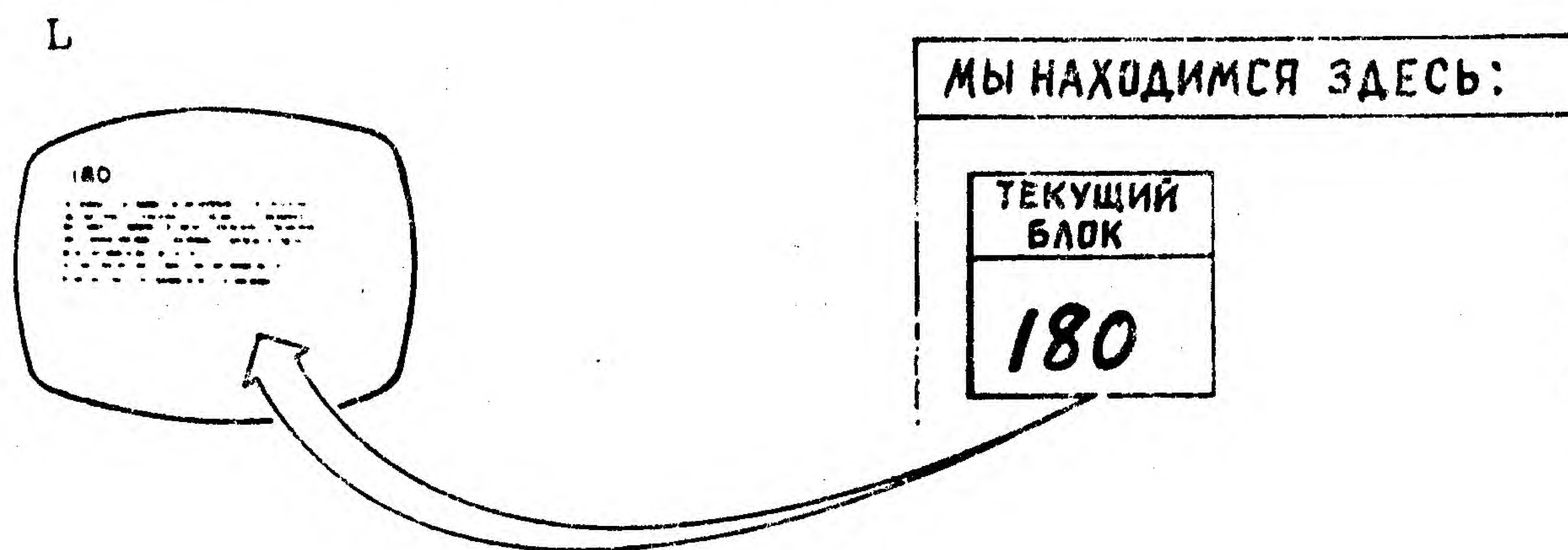
Сделав какой-то блок «текущим», вы можете распечатывать его, просто набирая слово

```
L
```

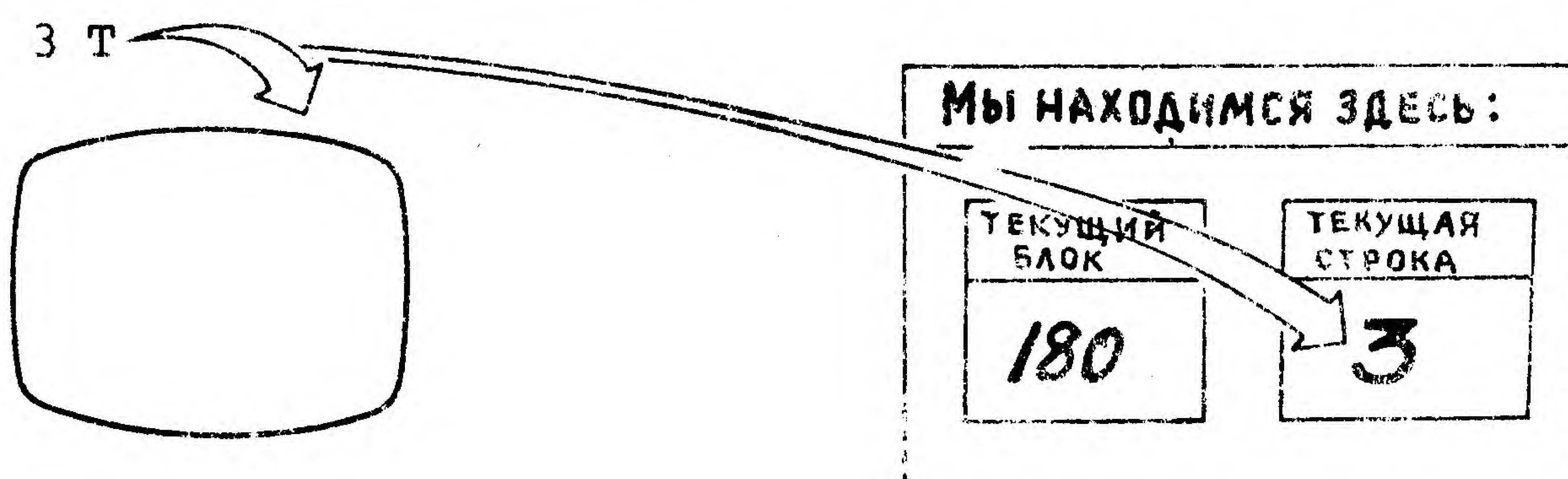
В отличие от **LIST** для **L** не нужно указывать номер блока, так как это слово распечатывает текущий блок.

Подготовив блок с помощью **LIST** к редактированию, неплохо было бы выполнить команду **WIPE** (СТЕРЕТЬ). Иногда никем не используемый блок, на первый взгляд пустой, содержит разного

рода символы, что может воспрепятствовать его загрузке. Слово **WIPE** заполняет текущий блок пробелами, после чего тот становится действительно пустым.



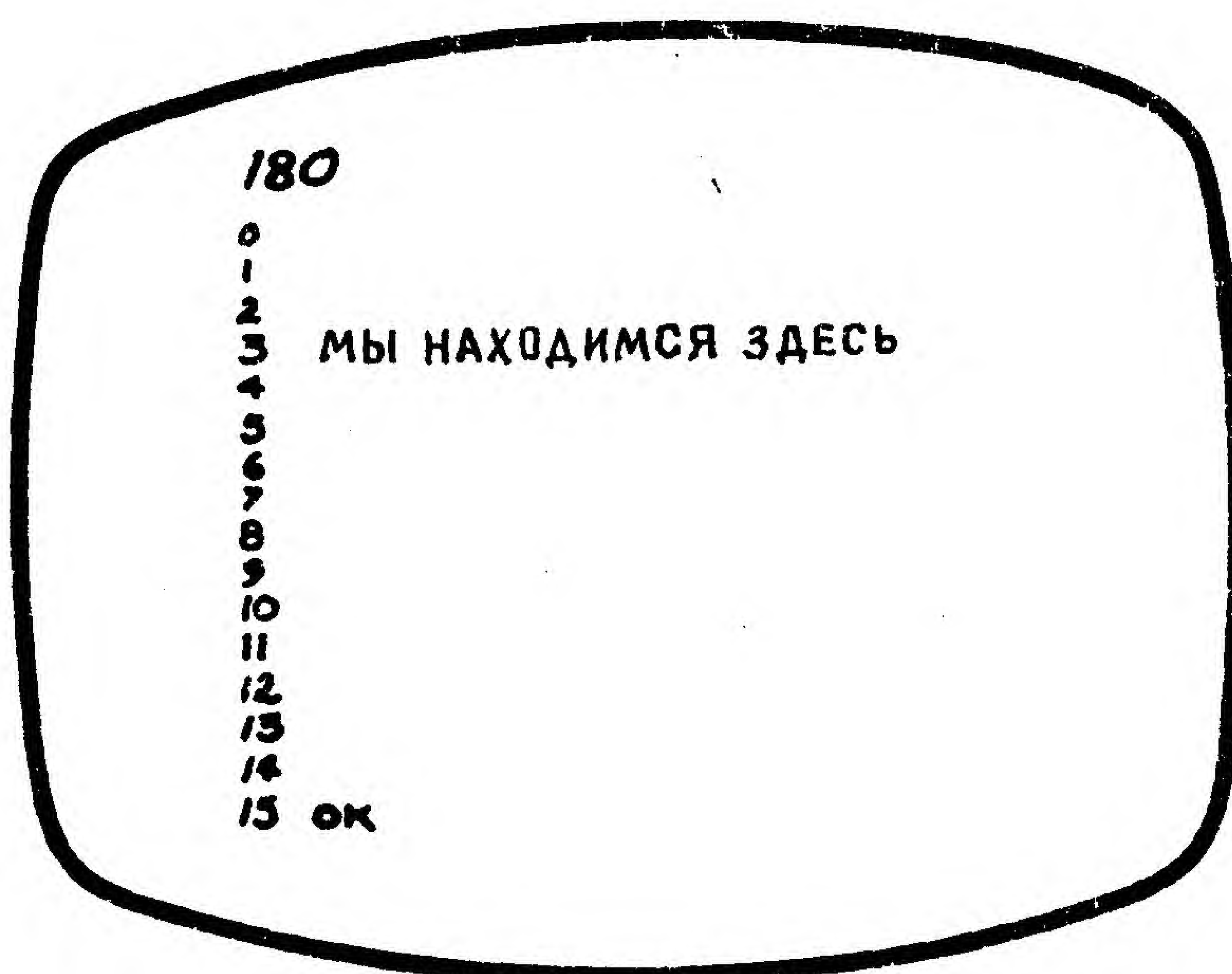
Поскольку у нас есть текущий блок, выберем и текущую строку с помощью слова **T**. Предположим, вы должны записать какую-то информацию в строку 3[1]. Наберите на клавиатуре



Слово **T** выделяет выбранную строку, заменяя ее изображение на негативное. Теперь, после того, как вы зафиксировали место, куда будете вносить исправления, можно в текущую строку занести некоторый текст с помощью команды **P** (**PUT** — ВСТАВИТЬ):

```
P МЫ НАХОДИМСЯ ЗДЕСЬ<return>
```

P — *вставляет* текст, следующий за этой командой (вплоть до символа возврата каретки), в текущую строку.



[1] *Для любознательных.* На самом деле в качестве указателя выступает не номер строки, а позиция курсора. Более подробно это будет показано в следующих сносках.

Помните, что текущая позиция, установленная вами, остается прежней, так что если вы сейчас наберете

P А ТЕПЕРЬ МЫ ЗДЕСЬ<return>

то увидите, что в строке 3 прежний текст заменен на более поздний.

Аналогично если вы введете **P**, а затем по меньшей мере два пробела (один — чтобы отделить **P** от текста, а другой — в качестве вводимого текста), то на месте прежней строки будет строка из одних пробелов. Другими словами, в этом случае очищается строка. В данной главе символ ƀ означает пробел, так что для заполнения строки пробелами нужно набрать на клавиатуре

Pьь<return>

КОМАНДЫ СИМВОЛЬНОГО РЕДАКТИРОВАНИЯ

Здесь мы рассмотрим, как вставлять и удалять текст в пределах строки.

F

Прежде чем вставлять или удалять текст, вы должны суметь подвести курсор *редактора* (не путать с курсором Форта) к месту вставки или удаления. Наш редактор обозначает позицию курсора тем, что дает следующий за ним текст до конца строки в негативном изображении. Предположим, что текущее содержимое строки 3 таково:

ЧЕМ ЛУЧШЕ КОМПЬЮТЕР, ТЕМ БОЛЬШЕ ПАЯТЬ

и вам нужно вставить недостающую букву М в слово ПАМЯТЬ. Курсор находится в начале строки. Для того чтобы переместить его за ПА, введите команду **F** (FIND — НАЙТИ) с фрагментом ПА:

F ПА<return>

Слово **F** будет просматривать текст, начиная с текущей позиции курсора, до тех пор, пока не *найдет* заданный фрагмент (в нашем случае ПА), после чего переместит курсор за найденный фрагмент.



ЧЕМ ЛУЧШЕ КОМПЬЮТЕР, ТЕМ БОЛЬШЕ ПАЯТЬ



ЧЕМ ЛУЧШЕ КОМПЬЮТЕР, ТЕМ БОЛЬШЕ ПАЯТЬ



ЧЕМ ЛУЧШЕ КОМПЬЮТЕР, ТЕМ БОЛЬШЕ ПА|ЯТЬ|

I

Так как курсор подведен к требуемому месту, достаточно ввести

```
I M<return>
```

и **I** *вставит* (INSERT — ВСТАВИТЬ) символ М за курсором.

```
ЧЕМ ЛУЧШЕ КОМПЬЮТЕР, ТЕМ БОЛЬШЕ ПАМЯТЬ
```

E

Чтобы удалить фрагмент с помощью команды **E** (ERASE — СТЕРЕТЬ), вы должны его сначала найти, послав команду **F**. Например, если вы хотите удалить слово ЛУЧШЕ, в первую очередь восстановите положение курсора:

```
3 T<return>
```

после чего введите

```
F ЛУЧШЕ<return>
```

```
ЧЕМ ЛУЧШЕ КОМПЬЮТЕР, ТЕМ БОЛЬШЕ ПАМЯТЬ
```

и далее

```
E<return>
```

Слово **E** удалит фрагмент, который вы только что задали в команде **F**:

```
ЧЕМ ЛУЧШЕ КОМПЬЮТЕР, ТЕМ БОЛЬШЕ ПАМЯТЬ
```

После этого **E** выведет исправленную строку:

```
ЧЕМ КОМПЬЮТЕР, ТЕМ БОЛЬШЕ ПАМЯТЬ
```

Курсор указывает место, куда вы можете вставить другое слово:

```
I МОЩНЕЕ<return>
```

```
ЧЕМ МОЩНЕЕ КОМПЬЮТЕР, ТЕМ БОЛЬШЕ ПАМЯТЬ
```

D

По команде **D** (DELETE — УДАЛИТЬ) находится и *удаляется* заданный фрагмент. В ней фактически сочетаются две команды:

F и **E**. Например, если ваш курсор находится в таком положении:

```
ЧЕМ МОЩНЕЕ |КОМПЬЮТЕР, ТЕМ БОЛЬШЕ ПАМЯТЬ|
```

то вы можете удалить слово КОМПЬЮТЕР, набрав

```
D КОМПЬЮТЕР<return>

ЧЕМ МОЩНЕЕ|, ТЕМ БОЛЬШЕ ПАМЯТЬ|
```

Однако вы снова можете вставить фрагмент текста в то место строки, на которое указывает сейчас курсор:

```
I ГОЛОВА<return>

ЧЕМ МОЩНЕЕ ГОЛОВА|, ТЕМ БОЛЬШЕ ПАМЯТЬ|
```

Применение команды **D** чревато ошибками более, чем последовательности команд **F** и **D**, так как при двухшаговом способе вы сначала четко указываете, что нужно удалить, а затем удаляете.

R

По команде **R** (REPLACE — ЗАМЕНИТЬ) заменяется фрагмент текста, который вы только что нашли. Эта команда объединяет в себе команды **E** и **I**. Например, если курсор показывает на фрагмент

```
|КОМПЬЮТЕРУ НУЖЕН ХОРОШИЙ ТЕРМИНАЛ      |
```

и вы ввели

```
F У НУЖЕН<return>

R САМ<return>
```

то получите следующее:

```
КОМПЬЮТЕР САМ |ХОРОШИЙ ТЕРМИНАЛ|
```

Команду **R** нужно применять в тех случаях, когда требуется сделать вставку *перед* определенным фрагментом текста. Например, если вы в нулевой строке пропустили символ **E**:

```
|( Учебные определения)                 MPTY|
```

то не так просто с помощью **F** найти для этой буквы место. Вы должны провести курсор через множество пробелов к требуемому (перед МРТУ). В данной ситуации лучше воспользоваться таким приемом:

```
F MPTY<return>
```

затем

```
R EMPTY<return>
```

TILL (ДО)

Самой мощной командой удаления является **TILL**. Она удаляет все, начиная с текущего положения курсора *до* указанного фрагмента включительно. Например, после применения к строке

```
ПРОГРАММИРУЙТЕ НА|ЧЕМ УГОДНО, ТОЛЬКО НЕ НА ФОРТЕ!
```

(заметьте, где находится курсор) команды **TILL**

```
TILL НА<return>
```

останется лишь текст:

```
ПРОГРАММИРУЙТЕ НА |ФОРТЕ!
```

Эта фраза звучит приятнее, не правда ли?

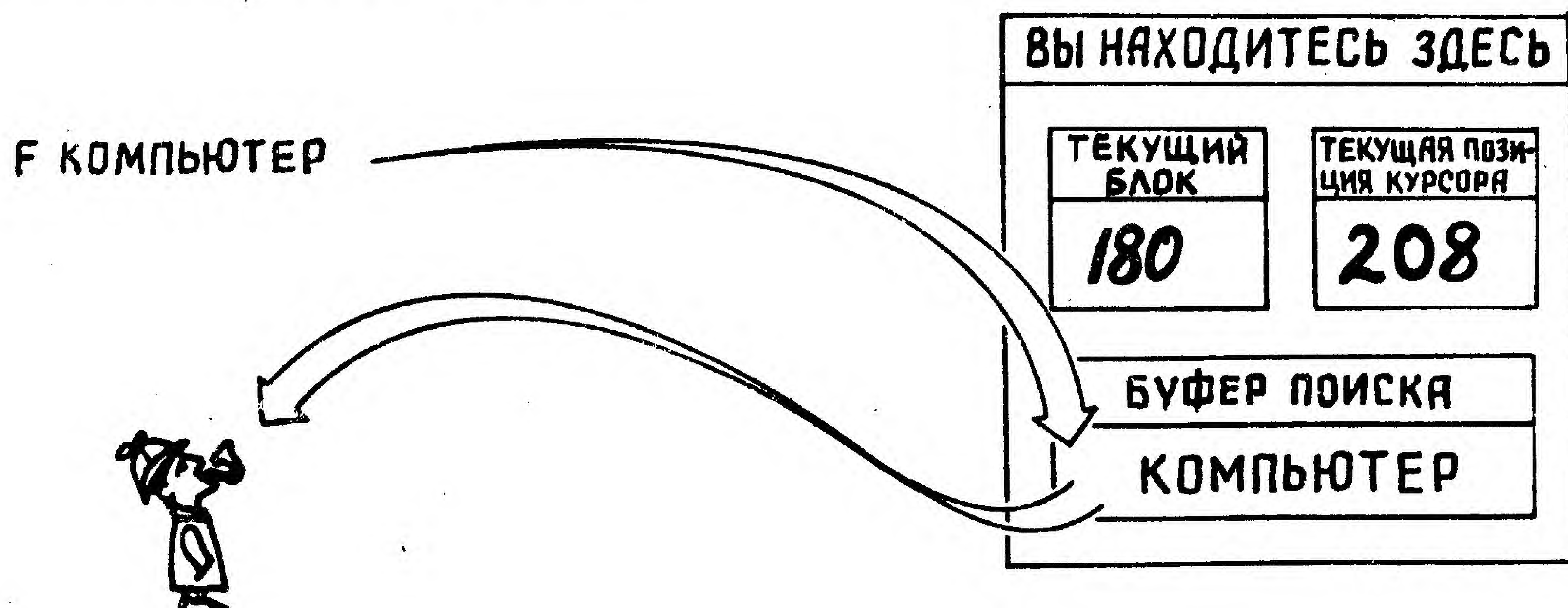
С помощью **TILL** осуществляется поиск в пределах текущей строки, а не по всей оставшейся части блока.

БУФЕР ПОИСКА И БУФЕР ВСТАВОК

Для того чтобы использовать редактор эффективно, вы должны разобраться в том, как работают его буферы поиска и вставок. Вы можете и не знать, что когда вы набираете на клавиатуре

```
F КОМПЬЮТЕР<return>
```

то по команде **F** фрагмент КОМПЬЮТЕР прежде всего помещается в так называемый буфер поиска. С точки зрения компьютерной терминологии буфер — это участок памяти для временного размещения данных. Буфер поиска находится в оперативной памяти компьютера (ОЗУ).

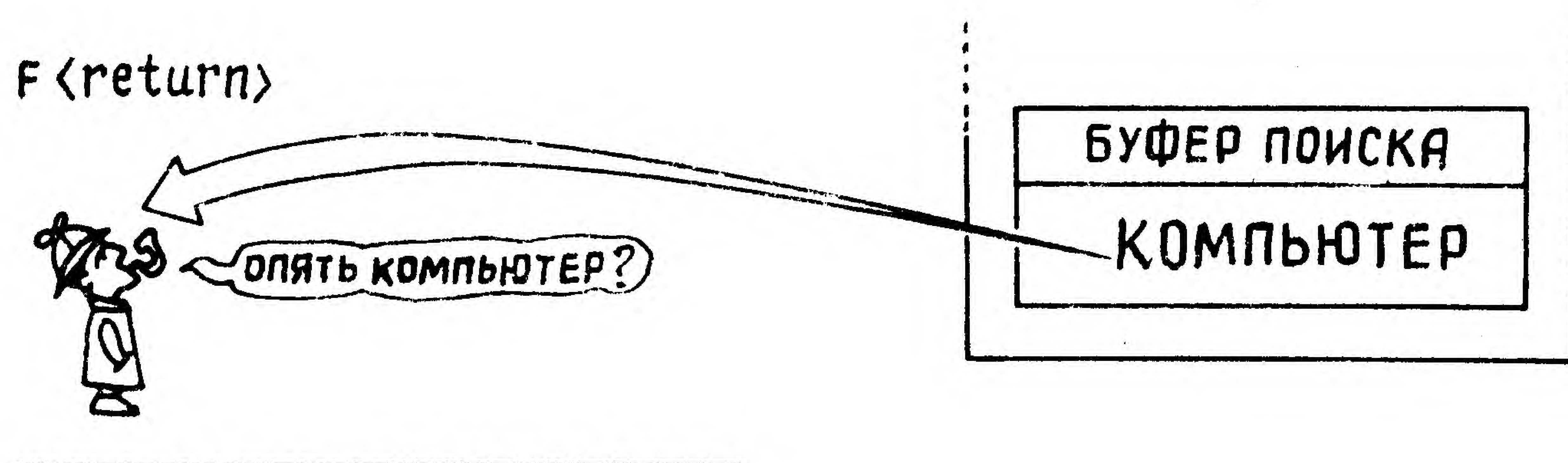


Команда **F** инициирует поиск такого фрагмента в строке, который соответствует содержимому буфера поиска.

Теперь вам понятно, что происходит, когда вы используете команду **F** следующим образом:

```
F<return>
```

т. е. в том случае, когда за **F** непосредственно следует символ возврата каретки. В строке осуществляется поиск такого фрагмента, который уже находится в буфере поиска как результат последнего выполнения команды **F**



Это позволяет находить многочисленные «вхождения» одного и того же фрагмента текста без его повторного набора. Например, предположим, что в строке 8 содержится следующее утверждение:

```
И СМЕХ, И СЛЕЗЫ, И ЛЮБОВЬ
```

Курсор указывает начало строки, и вы хотите удалить последний союз И. Введите следующую фразу:

```
F Иь<return>

И|СМЕХ, И СЛЕЗЫ, И ЛЮБОВЬ
```

Теперь фрагмент И находится в буфере поиска и вы можете просто несколько раз подряд набрать команду **F**:

```
F<return>
F<return>
```

и т. д. до тех пор, пока не доберетесь до того вхождения И, которое вам нужно и вы можете его удалить с помощью команды **E**. (По команде **E** подсчитывается число символов, находящихся в буфере поиска, и удаляется такое же число символов, предшествующих курсору.) Между прочим если вы попытались ввести команду **F** еще раз, то вы бы получили следующий ответ:

```
F И none
```

что означает: «В данной строке вхождений И нет». Иными словами, команда **F** не нашла в строке фрагмента, соответствующего содержимому буфера поиска и поэтому возвратила вам слово И с сообщением об ошибке NONE.

Как уже отмечалось выше, команда **D** сочетает в себе две команды **F** и **E**, поэтому она также использует буфер поиска. Поместив курсор в начало строки

```
И СМЕХ, И СЛЕЗЫ, И ЛЮБОВЬ
```

и фрагмент И в буфер поиска, вы можете удалить все вхождения И, введя несколько раз команду **D**:

```
D<return>
D<return>
D<return>
смех, слезы |любовь
```

Буфер вставок используется командой **I**. Для того чтобы вставить содержимое этого буфера в то место, куда указывает курсор, достаточно просто набрать на клавиатуре

```
I<return>
```

Рассмотрим пример, в котором показано, как можно одновременно применять буфер поиска и буфер вставок. Предположим, что в какой-то строке содержится следующая информация:

```
Я ЖИВУ, Я ЛЮБЛЮ, Я СТРАДАЮ
```

Теперь подведем курсор

```
F Яь<return>
Я |ЖИВУ, Я ЛЮБЛЮ, Я СТРАДАЮ
```

и осуществим вставку

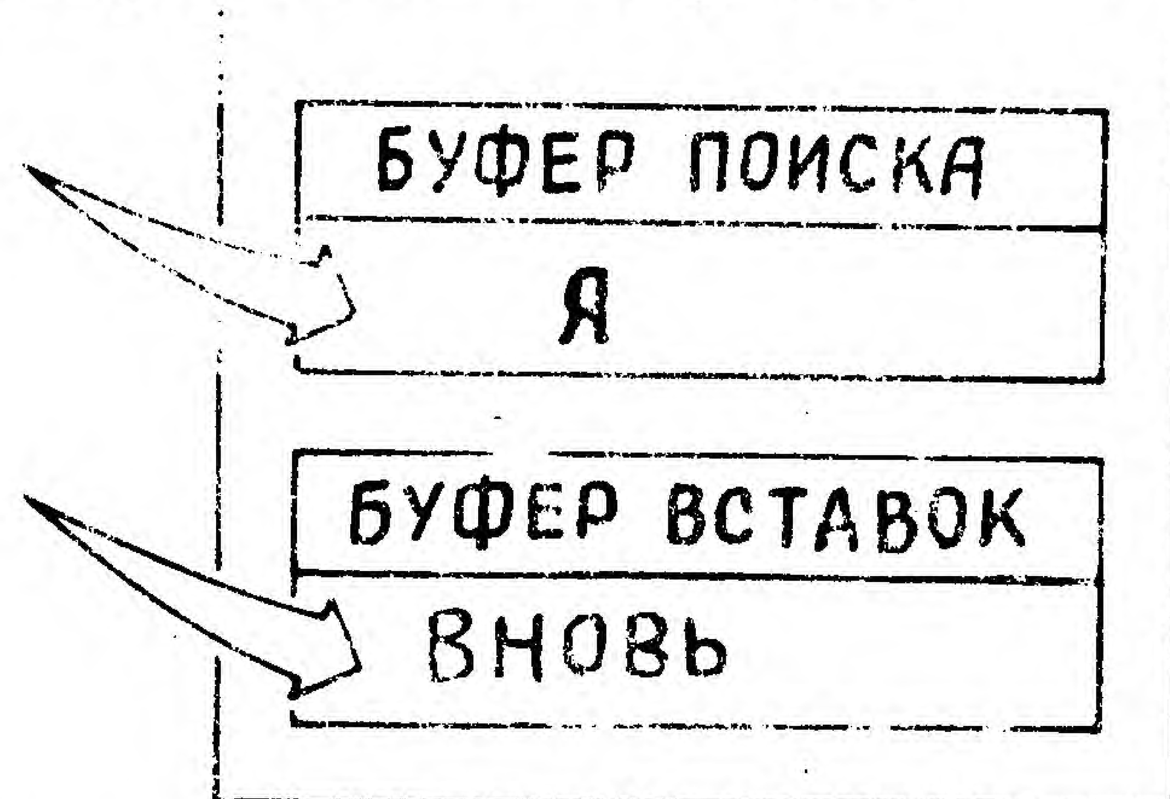
```
I ВНОВЬ<return>
Я ВНОВЬ |ЖИВУ,Я ЛЮБЛЮ,Я СТРАДАЮ
```

---

[1] *Для любознательных.* Редактору достаточно хранить только позицию курсора, а не указатель текущей строки. Так как в строке помещается 64 символа, редактор с помощью слова **/MOD** всегда сможет вычислить положение курсора, скажем, для 16-го символа в строке 3:

```
208 64 /MOD . . 3 16 ok
```

В результате получим



Наберите на клавиатуре

```
F<return>
I<return>
F<return>
I<return>
```

```
Я ВНОВЬ ЛЮБЛЮ, Я ВНОВЬ ЖИВУ, Я ВНОВЬ СТРАДАЮ
```

КОМАНДЫ РЕДАКТИРОВАНИЯ СТРОК

Итак, вы уже знаете способы изменения символов и слов и можете приступить к корректированию целых строк.

P

Слово **P**, которое мы уже ввели ранее, использует тот же буфер вставок, что и команда **I**. Предположим, что в вашем буфере вставок все еще находится фрагмент ВНОВЬ из предыдущего примера, а строка 14 все еще является текущей. Наберите на клавиатуре следующий текст:

```
P<return>
```

В результате прежнее содержимое строки 14 заменяется содержимым буфера вставок и в ней теперь находится только одно слово ВНОВЬ.

Чтобы получить представление об этой команде, посмотрите три примера ее применения:

```
1.  P ВЕСЬ ЭТОТ ТЕКСТ<return>

2.  Pьь<return>

3.  P<return>
```

В первом примере указанная строка помещается в буфер вставок, а затем в текущую строку, во втором заполняется пробелами буфер вставок, а затем и текущая строка, и, наконец, в третьем содержимое буфера вставок вносится в текущую строку.

U

Подобную функцию выполняет и слово **U**. Оно помещает содержимое буфера вставок *ниже* текущей строки. К примеру, допустим, что ваш блок имеет вид:

```
1. АДАМС
2. БРАУН
3. КЬЮДАХИ
4. ДЭВИС
5. ЭЛМЕР
6.
7.
```

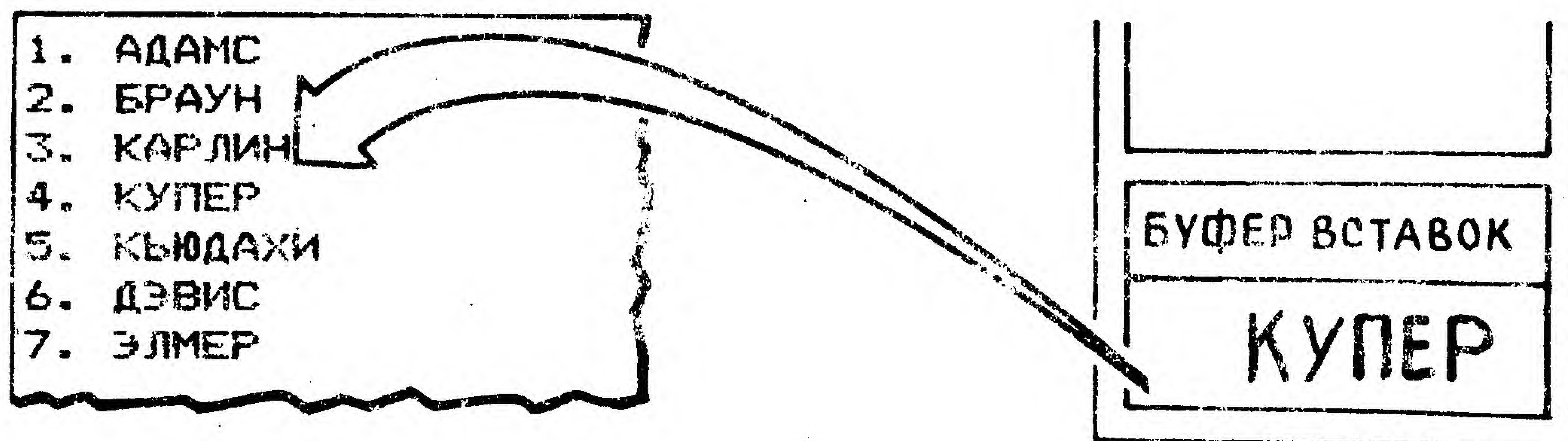
Если вы передвинете курсор ко второй строке:

```
2 T
```

а затем наберете

```
U КАРЛИН<return> ok
U КУПЕР<return> ok
```

то получите следующее:



Вместо того чтобы заменить текущую строку, команда **U** «втискивает» содержимое буфера вставок между текущей и последующими строками, передвигая их ближе к концу. Если бы в строке 15 находилась какая-то информация, она была бы вытеснена за пределы экрана и потеряна. Когда вы добавляете последовательность строк, проще иметь дело с командой **U**, а не **P**, например:

```
1 T P АДАМС<return> ok
U БРАУН<return> ok
U КЬЮДАХИ<return> ok
U ДЭВИС<return> ok
```

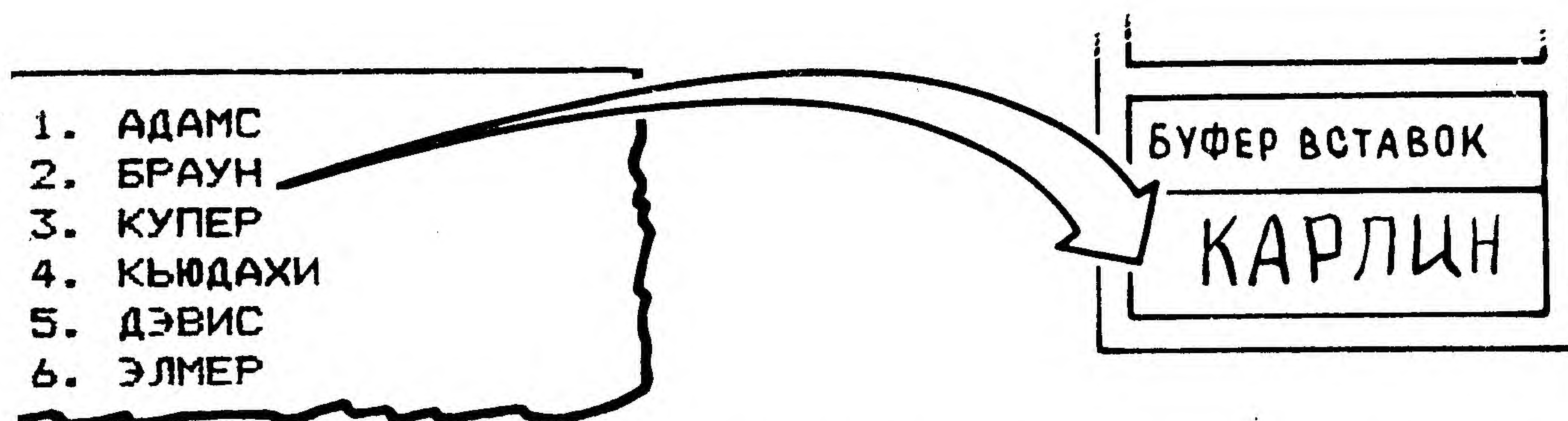
Перечисленные выше три способа использования команды **P** применимы также и к команде **U**.

X

Команда **X** по своему действию противоположна команде **U**. Она *извлекает* текущую строку. Если в рассмотренном выше примере мы сделаем строку 3 текущей (с помощью предложения 3T), а затем введем

```
X<return>
```

то строка 3 будет удалена, а нижние строки передвинутся вверх:



Как видите, по команде **X** извлеченная строка тоже помещается в буфер вставок. Это облегчает ее перемещение в дальнейшем. Например, последовательно вводя два предложения

```
9 T<return>
```

и

```
P<return>
```

вы можете поместить КАРЛИН в строку 9.

Для того чтобы вставить новую нулевую строку, а прежнюю нулевую строку опустить ниже, нужно сначала поместить эту новую строку *под* нулевой:

```
0 T U ЭТО НОВАЯ НУЛЕВАЯ СТРОКА.<reurn>
```

а затем первые две строки поменять местами:

```
0 T X U<return>
```

КОМБИНИРОВАННЫЕ КОМАНДЫ РЕДАКТИРОВАНИЯ

N и **B**

Вводя слово **N**, вы тем самым добавляете единицу к номеру текущего блока. Таким образом, очередная набранная комбинация символов

```
N L
```

вызовет печать *следующего* блока. Аналогично слово **B**





уменьшит на единицу номер текущего блока. Иными словами, комбинация

```
B L
```

позволяет вам вывести на печать предыдущий блок.

В некоторых системах **N** объединена с командой **L**, так что, применяя **N**, мы переходим к следующему блоку и выводим его содержимое. Вы можете при необходимости переопределить команды **N** и **B** с тем, чтобы они выполнялись так же — Форт допускает переопределение команд применительно к *вашим* потребностям.

COPY

Слово **COPY** позволяет *копировать* содержимое одного блока в другой. При этом вытесняются любые данные, находившиеся в приемном блоке до копирования. Эта команда имеет следующий формат:

```
откуда куда COPY
```

Если, к примеру, вы введете

```
153 200 COPY
```

то информация, находящаяся в блоке 153, будет скопирована в блок 200. Пусть у вас войдет в привычку вводить **FLUSH** после каждого выполнения **COPY**.

S

Команда **S** представляет собой расширенную версию команды **F**. Она дает возможность осуществлять *поиск* указанного текста в пределах текущего блока, а затем и в пределах последующих блоков вплоть до того блока, который вы указали в команде. Например, если текущим является блок 180 и вы ввели предложение

```
185 S СОКРОВИЩА
```

то **S** инициирует поиск «сокровища» в блоках начиная с 180-го и заканчивая 184-м. Если искомый фрагмент обнаруживается, то на экран выводится блок, содержащий данный фрагмент, а курсор устанавливается в его начале.

Слово **S** оставляет в стеке номер блока, на котором закончился поиск, так что если вам требуется продолжить поиск того же самого фрагмента, просто введите команду **S**.

G и BRING (ПЕРЕНОС)

Слово **B** берет одну строку из другого блока и вставляет ее перед текущей строкой (перемещая вниз текущую и все последующие строки).

Пусть курсор указывает на строку 3 следующего блока:

```
1
2 Куда ж ты, милая
3 , девалась? И где искать тебя теперь?
```

Пропущенный текст находится в строке 10 блока 38. Наберите на клавиатуре

```
38 10 G<return>
```

и вы получите следующее:

```
1
2 Куда ж ты, милая
3 пропущенная строка
4 , девалась? И где искать тебя теперь?
```

Слово **BRING** перемещает сразу группу строк. Выражение

```
38 10 14 BRING
```

переместит из блока 38 строки 10—14.

M

Некоторые разработчики Форта вместо команд **G** и **BRING** используют команду **M**, которая инициирует действия, противоположные вызываемым командой **G**. Она перемещает текущую строку в заданную строку экрана.

```
190 2 M<return>
```

K

Слово **K** меняет местами содержимое буферов поиска и вставок. Это полезно в тех ситуациях, когда вы случайно командой **D** удалили нужный вам текст. Так как удаленный текст находится в буфере поиска, вам достаточно ввести команду **K**, которая переместит его в буфер вставок, а затем команду **I**.

Можно выполнить и обратное действие. Если вы ошибочно вставили некоторый фрагмент не на свое место, переместите его посредством **K** в буфер поиска, а затем удалите с помощью **E**.

Попытайтесь самостоятельно поменять местами два слова в одной и той же строке, используя слово **K**.

В ряде систем вместо того, чтобы нажимать клавишу RETURN, вы можете с помощью символа возврата каретки обозначать конец фрагмента текста и таким образом в одной строке вводить более одной команды. Например, вы можете ввести

```
D ФРУКТЫ^ I ОРЕХИ<return>
```

Весь текст разместился в одной строке, а результат вы получите такой же, как если бы вы набрали на клавиатуре следующее:

```
D ФРУКТЫ<return>
```

и

```
I ОРЕХИ<returns>
```

Итак, мы рассмотрели команды редактора. Поскольку Форт по своей природе — язык гибкий, а пользователи при необходимости могут определять собственные команды редактирования, набор команд редактора в вашей системе может отличаться от набора, описываемого в настоящей книге. В конце раздела приводятся все команды, о которых здесь шла речь.

В заключение следует отметить, что редактор применительно к Форту это не программа, как, возможно, принято в других языках, а, скорее, набор слов. Его часто называют словарем, но к вопросу о словарях мы вернемся позднее.

Определение местоположения исходного текста (полезный прием).

В некоторых Форт-системах имеется слово **LOCATE** (ОПРЕДЕЛИТЬ-МЕСТОПОЛОЖЕНИЕ) или **VIEW**. Если вы введете следующее предложение:

```
LOCATE РАЗМЕР-ЯИЦ
```

то получите распечатку текста блока, содержащего определение РАЗМЕР-ЯИЦ. При этом указанное слово должно быть загруженным, т. е. находиться в словаре в данный момент. (В отдельных системах вы можете находить местоположение выборочных системных определений и слов вашей прикладной программы, но вы не имеете права определять местоположение слов из предварительно скомпилированного участка.)

Ниже следует перечень слов Форта, приведенных в настоящей главе:

FORGET имя	(--)	С помощью этого слова мы забываем (удаляем из словаря) указанное слово и все слова, внесенные в словарь после него.
LIST	(n --)	Вывод на экран дискового блока.
LOAD	(n --)	Загрузка дискового блока (компиляция или выполнение). Блок 0 обычно загружен быть не может.
FLUSH	(--)	Запись всех обновленных дисковых буферов на диск, после чего освобождение этих буферов.
\	(--)	Пропуск оставшегося текста данной строки.
\S	(--)	Пропуск оставшегося текста экрана.
THRU	(нач кон --)	Загрузка всех блоков с номерами из диапазона от нач до кон .
.(текст)	(--)	Вывод текста сообщения, ограниченного правой круглой скобкой. Используется, как правило, за пределами определения через двоеточие.
THRIAD	(n --)	Вывод трех блоков с номерами, включающими n, начиная с номера, делящегося без остатка на 3.
SHOW	(нач кон --)	Вывод блоков с номерами из диапазона от нач до кон по три блока.
INDEX	(нач кон --)	Вывод комментария только для блоков, номера которых входят в диапазон от нач до кон .
LOCATE xxx или VIEW	(--)	Вывод содержимого блока, из которого было загружено определение слова xxx.

Команды редактирования - работа со строками

T	(n --)	Вывод заданной строки.
P Pьь или P xxx	(--)	Копирование заданного фрагмента, если есть, в буфер вставок, после чего помещение копии буфера вставок в текущую строку.
U Uьь или U xxx	(--)	Копирование заданной строки, если есть, в буфер вставок, после чего помещение копии буфера вставок в строку, следующую за текущей.
G	(блок строка)	Копирование заданной строки и помещение ее в строку перед текущей, со сдвигом текущей и всех последующих строк вниз.
BRING	(блок нач кон)	Получение строк в указанном диапазоне.
X	(--)	Копирование текущей строки в буфер вставок и извлечение этой строки из блока.
F или F xxx	(--)	Копирование указанной строки, если задана, в буфер поиска, после чего поиск данной строки в текущем блоке.

S или S xxx	(n --) или (n -- n)	Копирование указанной строки, если задана, в буфер поиска, после чего п р о с м о т р блоков от текущего до n-ного в поисках указанной строки. Если строка найдена, на стек помещается номер последнего просмотренного блока.
E	(--)	Используется следом за F. Удаляется столько символов перед курсором, сколько их в данный момент находится в буфере поиска.
D или D xxx	(--)	Копирование указанной строки, если задана, в буфер поиска, поиск очередного вхождения этого фрагмента в текущей строке и удаление его.
TILL или TILL xxx	(--)	Копирование указанной строки, если задана, в буфер поиска, после чего удаление всех символов, начиная от курсора и заканчивая последним символом заданной строки.
I или I xxx	(--)	Копирование указанного фрагмента, если задан, в буфер вставок, после чего помещение содержимого буфера вставок сразу же после курсора.
R или R xxx	(--)	Объединяются команды E и I. Замещение найденного фрагмента заданным фрагментом или содержимым буфера вставок.
↑ или ^	(--)	Отметка конца текста, помещаемого в буфер.

Комбинированные команды редактирования

WIPE	(--)	Заполнение текущего блока пробелами.
L	(--)	Вывод содержимого текущего блока.
N	(--)	Делается текущим следующий блок.
B	(--)	Делается текущим предыдущий блок.
COPY	(откуда куда --)	Копирование содержимое одного блока в другой.
K	(--)	Меняются местами содержимое буфера поиска и буфера вставок.

ОСНОВНЫЕ ТЕРМИНЫ

Блок. В Форте представляет собой раздел дисковой памяти, содержащий 1024 символа исходного текста.

Блок загрузки. Блок, который, будучи загруженным, сам осуществляет загрузку остальных блоков прикладной программы.

Буфер. Участок памяти для временного хранения данных.

Входной поток. Строка символов, предназначенная для обработки текстовым интерпретатором. Входной поток может подавать-

ся либо с текущего устройства ввода, например с клавиатуры (через буфер входного текста), либо с внешнего устройства, такого, как диск (через блочный буфер).

Исходный текст. Определение или несколько определений, написанных на языке, близком к естественному, с соблюдением правил пунктуации в противоположность к компилируемой форме представления определений, в которой те заносятся в словарь.

Нулевое определение. Определение, записываемое в форме

```
: имя ;
```

и не предусматривающее выполнение каких-либо операций. Оно служит как бы «закладкой» в словаре, отмечая место, до которого весь текст «забывается» по команде **FORGET.**

Оверлейный фрагмент. Фрагмент прикладной программы, замещающий при загрузке другой фрагмент в словаре.

Указатель. Участок памяти, куда может быть помещено число (или где оно может быть изменено), предназначенное в дальнейшем для ссылок.

УПРАЖНЕНИЯ

3.1. а) Введите ваши определения слов ДАР, ДАРИТЕЛЬ и СПАСИБО из упр. 1.1 и 1.3 в блок, после чего загрузите и исполните слово СПАСИБО.

б) с помощью редактора измените имя в определении ДАРИТЕЛЬ, после чего загрузите и исполните слово СПАСИБО снова. Что произойдет в этом случае?

3.2. Попытайтесь загрузить некоторые из наших математических определений из гл. 2 в какой-нибудь доступный блок, а затем загрузите его. Поэкспериментируйте.

ЛИТЕРАТУРА

[1] Brodie, Leo, *Thinking Forth* (Englewood Cliffs, N.J.: Prentice-Hall, 1984).

Глава 4
КОМПЬЮТЕР «ПРИНИМАЕТ РЕШЕНИЯ»

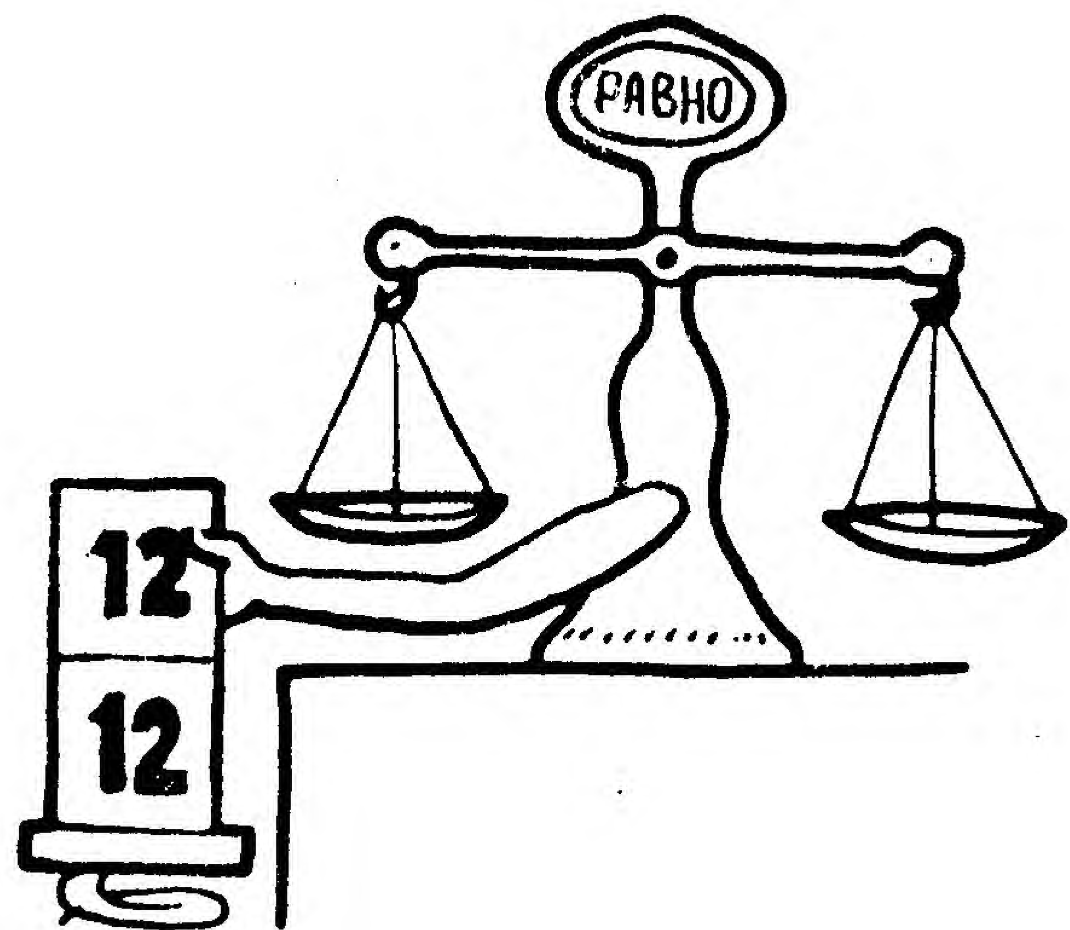
В этой главе вы научитесь писать программы таким образом, чтобы компьютер мог «принимать решения». Теперь вы будете воспринимать свой компьютер как довольно сложное устройство, а не просто как обычный калькулятор.

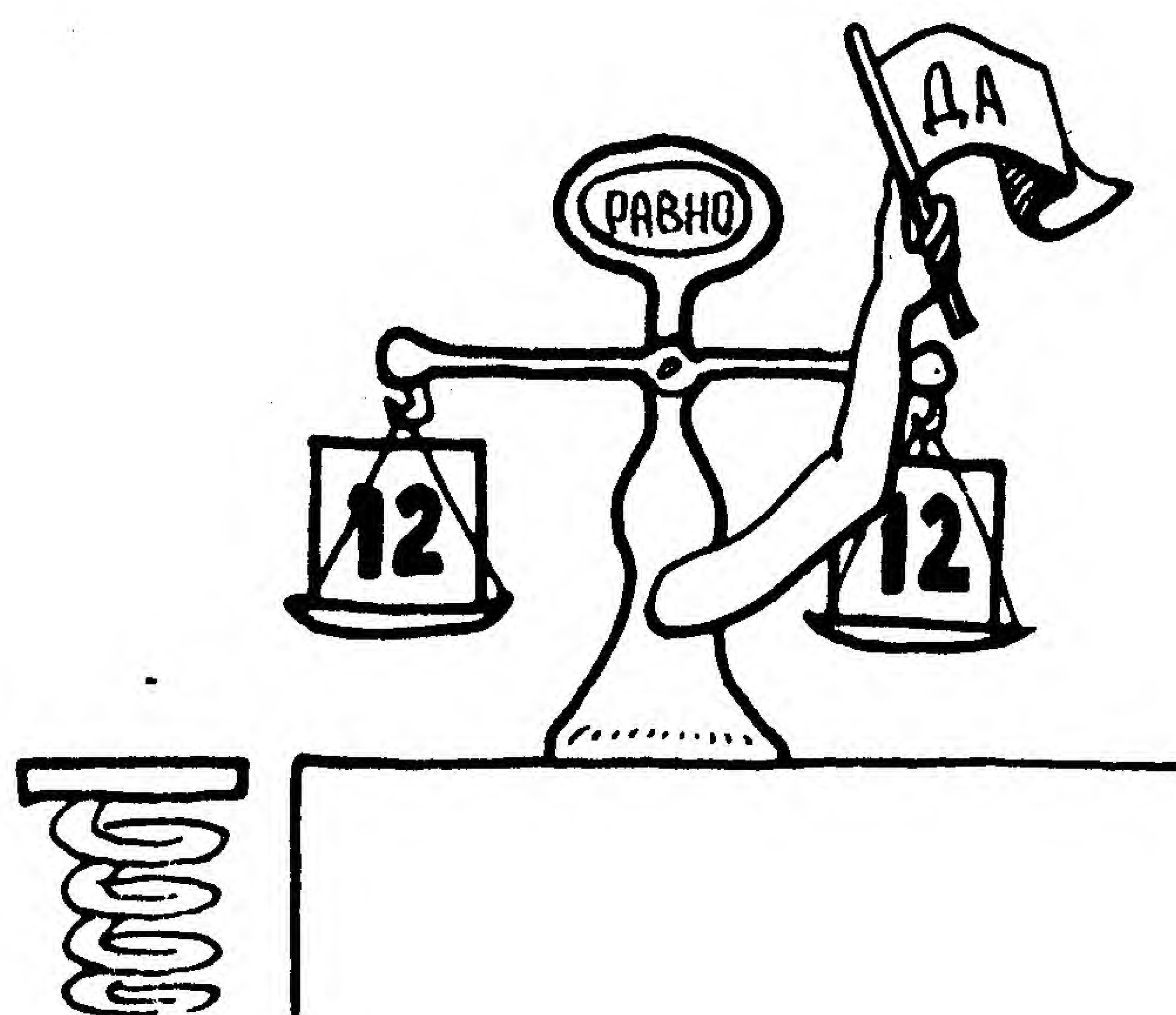
УСЛОВНЫЙ ОПЕРАТОР

Запишем на Форте простой оператор принятия решения. Допустим, вы программируете механический упаковщик яиц в картонные коробки. Некоторое механическое устройство уже подсчитало число яиц, находящихся в ленте конвейера, и поместило это число в стек. В соответствии со следующим предложением на Форте:

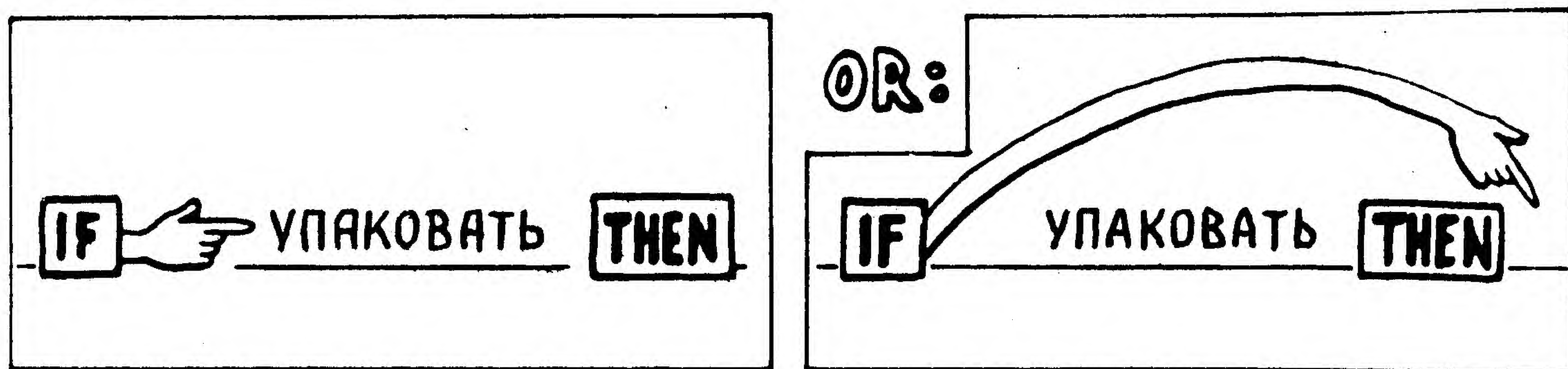
```
12 =  IF УПАКОВАТЬ THEN
```

осуществляется проверка: *равно* ли число в стеке 12, и ЕСЛИ (IF) да, ТО (THEN) выполняется слово УПАКОВАТЬ. Если число не равно 12, то выполняются слова, которые следуют за **THEN**.





Слово = выбирает два значения из стека и сравнивает их, чтобы проверить, равны ли они.



Если условие истинно (да), то будут выполнены слова, следующие за **IF**, если ложно — то слова, следующие за **THEN.**

Попытаемся выполнить этот оператор. Определим, например, следующее слово:

```
: ?ПОЛНА  ( число-яиц )        12 =  IF ." КОРОБКА ПОЛНА  " THEN ; ok
11 ?ПОЛНА ok
12 ?ПОЛНА КОРОБКА ПОЛНА ok
```

Заметим, что оператор **IF ... THEN** должен содержаться только внутри определения (через двоеточие), так что вы не можете использовать его в режиме калькулятора.

Абстрагируйтесь от всех значений, которые заключены в словах **IF** (ЕСЛИ) и **THEN** (ТО) в естественном языке. В Форте они выражают то, что операторы, следующие за **IF**, выполняются лишь в случае, *если* условие истинно, а операторы, следующие за **THEN**, *всегда*. Это равносильно тому, что вы прикажете компьютеру после принятия решения (выполнять или не выполнять слова, следующие за **IF**, в зависимости от того, истинно условие или ложно) продолжить выполнение остальной части определения (следующей за **THEN**). В нашем примере после **THEN** находится единственное слово ;, которое означает конец определения.

Если вам удобно, сразу представляйте запись в *постфиксной форме*. Вместо традиционного применения выражения **IF**:

```
IF  ( условие )  THEN  ( действие )  ENDIF
```

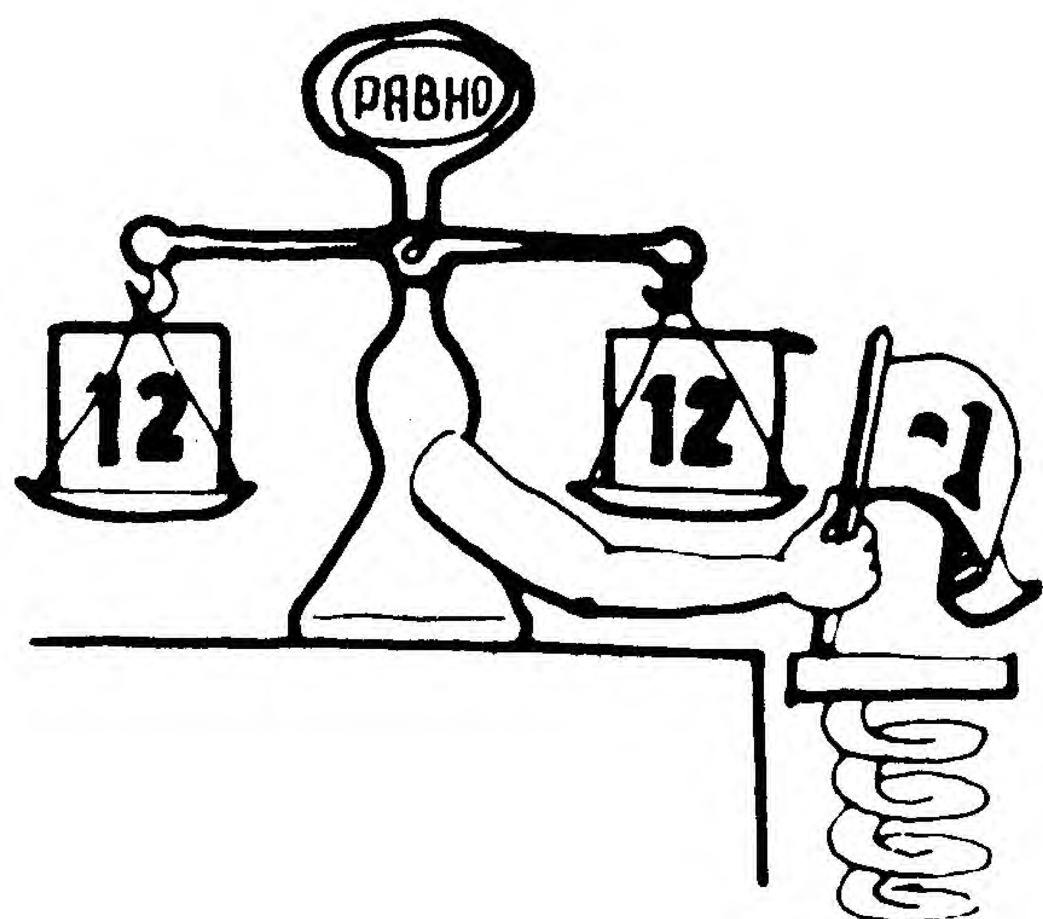
мы имеем

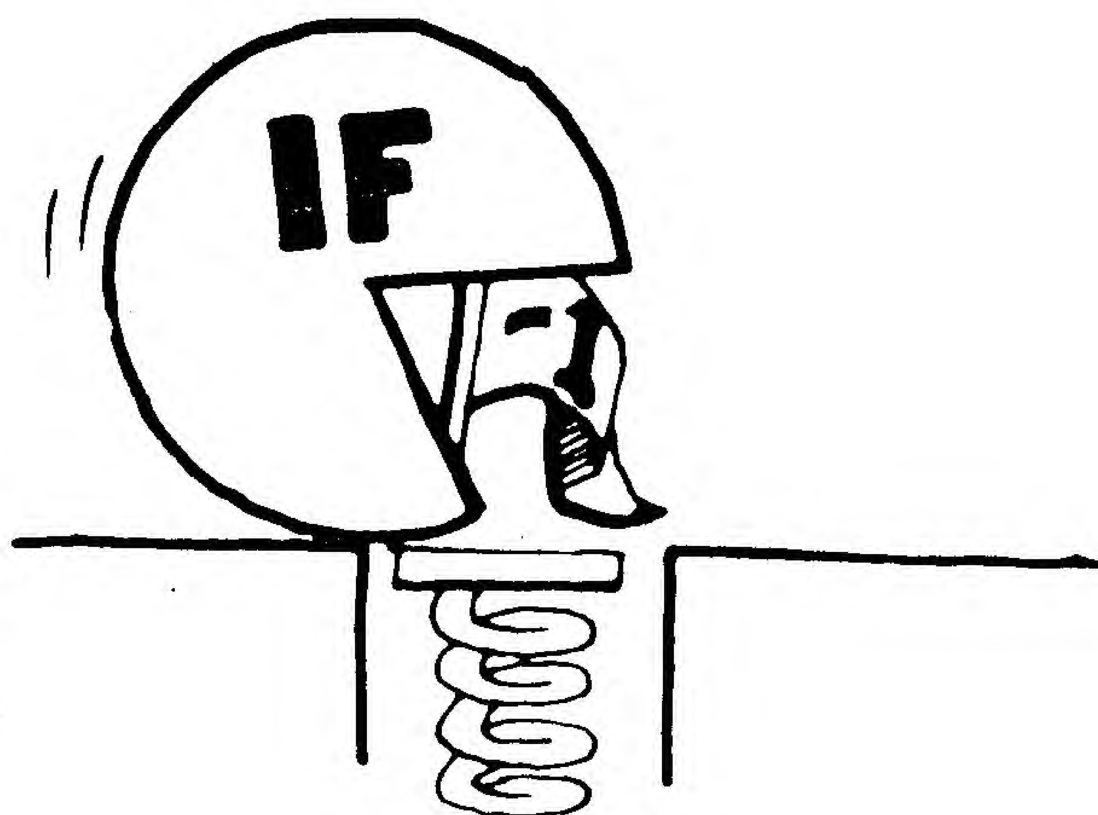
```
( условие )  IF  ( действие )  THEN
```

Помните, что каждому слову **IF** должно соответствовать свое **THEN**, причем в пределах одного и того же определения.

БОЛЕЕ ПОДРОБНО ОБ ОПЕРАТОРЕ IF

В результате выполнения операции сравнения флаг[1] на самом деле вверх не выбрасывается, как это показано на рисунках, а его значение *заносится в стек* подобно любому другому аргументу. Истина представляется —1 (отрицательной единицей), а ложь — 0 (нулем)[2]. Слово **IF** берет флаг из стека и использует его.





Попытайтесь ввести следующую фазу с терминала и пусть слово **.** выведет значение, представляющее флаг:

```
12 12 = . -1 ok          Да, 12 равно 12
11 12 = . 0 ok           Нет, 11 не равно 12
```

Можно вводить знаки операций сравнения непосредственно с вашего терминала, как в приведенном выше примере, но помните, что оператор **IF ... THEN** должен целиком находиться в пределах одного определения, поскольку его выполнение сопряжено с ветвлением программы. Слово **IF** будет воспринимать —1 как значение флага «истина», а 0 — как «ложь». Перед **IF** может стоять еще одно слово **NOT**[3], которое меняет значение флага в стеке на противоположное:

```
0 NOT . -1 ok
-1 NOT . 0 ok
```

Слово **NOT** позволяет изменить условие **IF** на обратное. Таким образом, мы можем записать

```
: ?ДВЕНАДЦАТЬ  ( n)  12 = NOT IF ." Не двенадцать. " THEN ;
```

---

[1] *Для начинающих.* На компьютерном жаргоне значение, которое один фрагмент программы оставляет другому в качестве сигнала, называется *флагом.*

[2] *Для пользователей систем, созданных до введения Стандарта-83.* В более ранних системах истина представлялась как 1.

[3] *Для систем фиг-Форт.* Используйте в этих целях 0=.

и если параметр n *не будет равен* 12, на печать будет выводиться фраза «Не двенадцать».

Использование стека в Форт-системе для передачи значений флага — одно из наиболее удачных решений, привлекательных прежде всего своей простотой. Вы можете, например, передать флаг в качестве аргумента другому слову (если нужно, то за пределы определения):

```
: .ДА?  ( ? -- )  IF  ." Да "  THEN ;
12 12 = .ДА? Да ok
```

(Знак вопроса в стековом комментарии означает флаг.) Можете ли вы назвать еще такой язык, который дает возможность записывать условие в одной процедуре, а оператор **IF** — в другой?

ОПЕРАЦИИ СРАВНЕНИЯ

Ниже приводится неполный список операций сравнения, которые вы можете применять перед выполнением оператора **IF ... THEN**[1].

```
=      равно
<>¹    не равно
<      меньше
>      больше
0=     равно-нулю
0<     меньше-нуля
0>     больше-нуля
```

Слова < и > требуют такого же расположения аргументов в стеке, как и арифметические операции:

ИНФИКСНАЯ ФОРМА	ПОСТФИКСНАЯ ФОРМА
2 < 10	2 10 <
17 > −39	17 −39 >

Рассмотрим другой пример. Проверим, не превышает ли температура лабораторного бойлера допустимого значения. Значение температуры нужно получить в стеке

```
: ?ЖАРКО  ( температура -- )
   220 > IF ." ОПАСНО!  Уменьшите нагрев " THEN ;
```

[1] *Для тех, у кого нет* <>. Используйте — (минус). См. разд. «Секреты оператора **IF**» в данной главе.

Если значение, находящееся в стеке *больше, чем* 220, то на терминал будет выведено сообщение об опасности. Вы можете выполнить это слово автономно, набрав на клавиатуре сначала определение, а затем и само слово. Перед словом вы должны набрать значение температуры:

```
290 ?ЖАРКО ОПАСНО!  Уменьшите нагрев ok
130 ?ЖАРКО ok
```

Вы можете проверить, является ли некоторое число нулем, отрицательным или положительным, с помощью трех следующих слов: **0 =**, **0 <**, **0 >**. Они эквивалентны выражениям **0 =**, **0 <**, **0 >** и отличаются лишь эффективностью предлагаемых операций.

АЛЬТЕРНАТИВНАЯ ВЕТВЬ УСЛОВНОГО ОПЕРАТОРА

Форт позволяет вам написать в рамках оператора **IF** с помощью слова **ELSE** (ИНАЧЕ) альтернативное выражение. В приведенном ниже примере дается определение, которое проверяет, является ли заданное число правильной календарной датой:

```
?ДЕНЬ   ( день)
  32 < IF ." Путь открыт " ELSE  ." Объезд " THEN ;
```

Если число в стеке меньше 32, то будет выдано сообщение «ПУТЬ ОТКРЫТ». В противном случае выдается сообщение «ОБЪЕЗД».



Представьте себе, что **IF** переключает стрелку железнодорожной колеи в зависимости от результатов проверки условия, после чего выполнение пойдет по одному из двух маршрутов, но в любом случае рельсы сойдутся у слова **THEN**. В компьютерной термино-

логии изменение путей выполнения операторов называется *ветвлением*[1].

Рассмотрим еще один пример. Как известно, деление любого числа на нуль невозможно, поэтому если вы попытаетесь выполнить эту операцию на каком-нибудь компьютере, то получите неправильный ответ.

Можно определить некоторое слово, выполняющее деление в том случае, если делитель не равен нулю[2]:

```
: /ПРОВЕРКА  ( числитель знаменатель -- результат )
   DUP 0= IF ." Знаменатель нуль " DROP  ELSE  /  THEN ;
```

Заметим, что сначала вы должны с помощью **DUP** создать копию знаменателя, так как выражение

```
0= IF
```

в процессе своего выполнения уничтожит его. Кроме того, слово **DROP** удаляет знаменатель, если деление не выполняется, так что независимо от того, будет ли выполняться деление, состояние стека в обоих случаях окажется одинаковым, т. е. независимо от того, выполнялась ли часть **IF** или **ELSE**, в стеке будет оставлен один аргумент. (Случай, когда указанные две части оставляют в стеке различное число аргументов, являются источником самых коварных ошибок: иногда при этом ваша программа работает, а иногда — нет.)

ВЛОЖЕННЫЕ КОНСТРУКЦИИ IF ... THEN

Имеется возможность помещать оператор **IF ... THEN** (или **IF ... ELSE ... THEN**) внутрь другого оператора **IF ... THEN**. Фактически вы можете создать оператор любой степени вложенности при условии, что каждый оператор **IF** будет иметь соответствующий оператор **THEN**.

---

[1] *Для специалистов*. В Форте нет оператора GOTO. Если вы не можете обойтись без этого оператора немного подождите. К концу книги вам станут ясны отрицательные последствия его буквального применения. Вместо GOTO мы предложим вам средства, позволяющие сделать ваши программы более корректными.

[2] *Для специалистов*. Как вы увидите ниже, существуют лучшие способы выполнения этих действий.

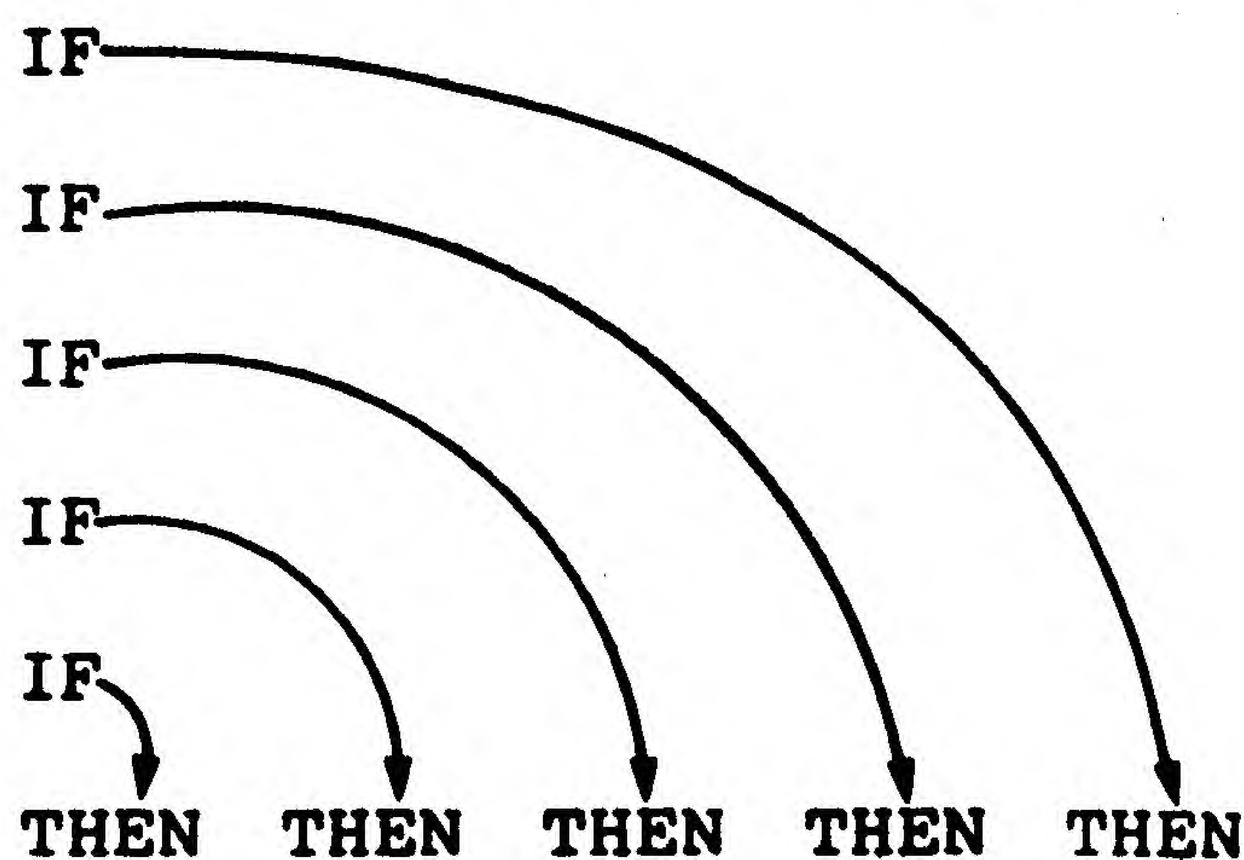
Рассмотрим следующее слово, которое определяет сорт яиц для продажи в зависимости от их размера (очень крупные, крупные и т. д.), по весу в унциях[1] на дюжину[2]:

```
: РАЗМЕР-ЯИЦ   ( унций-на-дюжину -- )
                DUP  18 < IF ." Брак "              ELSE
                DUP  21 < IF ." Мелкие "            ELSE
                DUP  24 < IF ." Средние "           ELSE
                DUP  27 < IF ." Крупные "           ELSE
                DUP  30 < IF ." Очень крупные "     ELSE
                             ." Ошибка "
                     THEN THEN THEN THEN THEN  DROP ;
```

Загрузив однажды слово РАЗМЕР-ЯИЦ, вы можете получать различные результаты, например[3]:

```
23 РАЗМЕР-ЯИЦ Средние ок
29 РАЗМЕР-ЯИЦ Очень крупные ок
40 РАЗМЕР-ЯИЦ Ошибка ок
```

Обратите внимание на некоторые особенности работы со словом РАЗМЕР-ЯИЦ. Все определение является совокупностью вложенных операторов **IF ... THEN**. Эти операторы вложены друг в друга, как матрешки. Пять слов **THEN**, расположенных внизу, соответствуют пяти словам **IF**, расположенным в обратном порядке:



Отметим также, что в конце определения должно быть слово **DROP**, которое позволит избавиться от лишнего исходного значения.

---

[1] 1 унция = 28,3 г. — *Примеч. пер.*

[2] Для *работающих за терминалом.* Поскольку приведенное определение уже достаточно сложное, рекомендуется вводить его из блока на диске.

[3] *Для любознательных.* Ниже приводится официальная таблица, согласно которой производится определение сорта яиц:

очень крупные	27—30	крупные	24—27
средние	21—24	мелкие	18—21

Можно ли удалить последний оператор **DUP** с тем, чтобы число убиралось из стека последней операцией сравнения, и оператор **DROP**? Нет, а почему?

Наше определение организовано таким образом, что оно легко воспринимается человеком. Большинство программирующих на Форте скорее смирятся с потерей памяти в блоке из-за пропусков между словами (блоков много), чем допустят то, чтобы их программы выглядели запутаннее, чем они есть на самом деле.

«СЕКРЕТ» ОПЕРАТОРА IF

Мы уже знаем, что **IF** принимает —1 за истину. Программирующие на Форте часто пользуются тем, что на самом деле **IF** воспринимает как истину *любое ненулевое значение* и только нуль как ложь[1]. Обычно вы над этим не задумываетесь, но бывают ситуации, когда такая реализация представляет интерес.

Так, если вы проверяете некоторое число только на равенство нулю, то операция сравнения вам не нужна. Определение рассмотренного нами слова /ПРОВЕРКА может поэтому принять более простую форму:

```
: /ПРОВЕРКА  ( числитель знаменатель -- результат)
    DUP IF  /  ELSE  ." Знаменатель нуль " DROP  THEN ;
```

Или, допустим, вам нужно узнать, является ли некоторое число в точности кратным десяти, например 10, 20, 30, 40 и т. д. Как известно, выражение

```
10 MOD
```

осуществляет деление на 10 и возвращает только остаток. А при вычислении кратного в остатке должен быть нуль, поэтому выражение

```
10 MOD 0=
```

соответствует флагу «истина» или «ложь».

---

[1] *Для сомневающихся.* Чтобы проверить это, введите следующий текст:

```
: ТЕСТ  ( ? )  IF ." Не нуль "  ELSE  ." Нуль "  THEN ;
```

Даже несмотря на то, что в данном определении нет операции сравнения, вы получите такой результат:

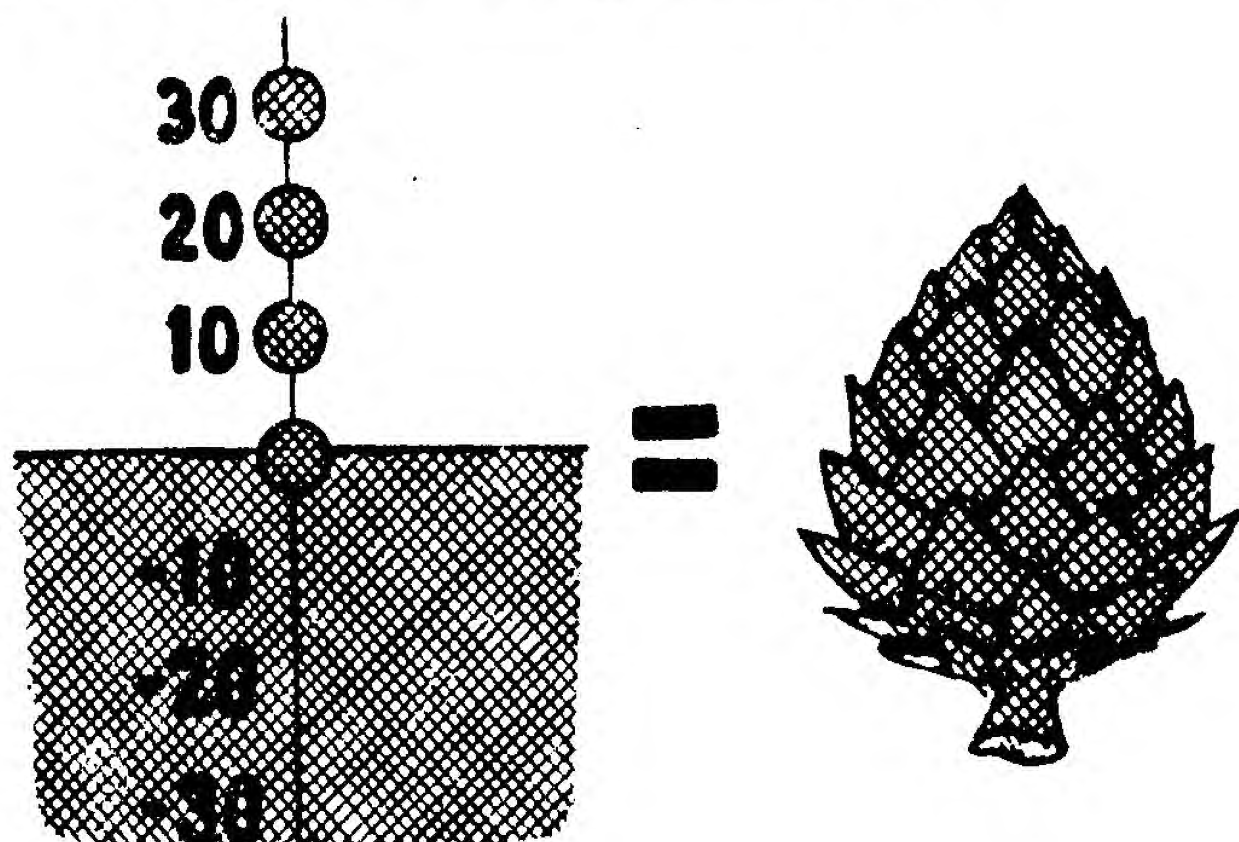
```
0 ТЕСТ Нуль ок
1 ТЕСТ Не нуль ок
8000 ТЕСТ Не нуль ок
```

Далее, слово — (минус) вы зачастую можете использовать в качестве операции сравнения, чтобы проверить на равенство два значения. Если вы выполняете операцию вычитания над двумя одинаковыми числами, то получаете нуль (ложь), но если при вычитании числа не равны, то вы получаете не нулевое значение (истину). В некоторых Форт-системах по этой причине отсутствует слово <>. Тем не менее <> не только более удобно, чем «не равно», но и оставляет на вершине стека в качестве истинного значения отрицательную единицу. Как вы увидите в гл. 7, операция сравнения имеет свои достоинства (в сравнении с арифметическим вычитанием).

НЕМНОГО ЛОГИКИ

Форт (как и большинство языков программирования) дает вам возможность комбинировать флаги. Возьмем, к примеру, комбинацию по принципу «или». Даны значения двух флагов. Если хотя бы одно из них истинно, то Форт-система выполнит действие, если оба значения ложны — не выполнит.

Поясним изложенное на следующем примере. Допустим, вы хотите вывести имя АРТИШОК в том случае, если исходное число *либо* отрицательно, *либо* кратно 10. Как это сделать на Форте? Рассмотрим выражение:




```
DUP 0<   SWAP   10 MOD   0=   OR
```

Ниже приводятся значения, которые получаются при исходном значении, скажем равном 30:

ОПЕРАТОР	СОДЕРЖАНИЕ СТЕКА		ОПЕРАЦИЯ
		30	
DUP	30	30	Содержимое стека (вершины) удваивается, поэтому мы можем проверять это значение дважды
0<	30	0	Значение отрицательно? Нет (нуль)
SWAP	0	30	Флаг и число меняются местами
10 MOD 0=	0	–1	Делится ли значение точно на 10? Да (отрицательная единица)
OR		–1	Флаги логически складываются

Флаги логически складываются? Что же происходит при таком сложении флагов? Вы получаете в результате истину, если хотя бы один из флагов (или сразу оба) истинны. Ниже приводятся четыре возможных сочетания двух флагов и результаты операции **OR** (ИЛИ) над ними.

1-й ФЛАГ	0	-1	0	-1
2-й ФЛАГ	OR 0	OR 0	OR -1	OR -1
РЕЗУЛЬТАТ	0	-1	-1	-1

Определение в нашем простом примере, следовательно, будет выглядеть так:

```
: ОВОЩ  ( n)  DUP 0<  SWAP  10 MOD 0= OR
   IF ." Артишок "  THEN ;
```

Ниже приводится улучшенный вариант приведенного выше определения слова ?ДЕНЬ. Предыдущее определение браковало только элементы, значение которых превышало 31, а отрицательные значения не допускались вовсе.

```
: ?ДЕНЬ  ( день)  DUP 1 <  SWAP 31 >  OR
     IF  ." Ошибка "  ELSE  ." Спасибо "  THEN ;
```

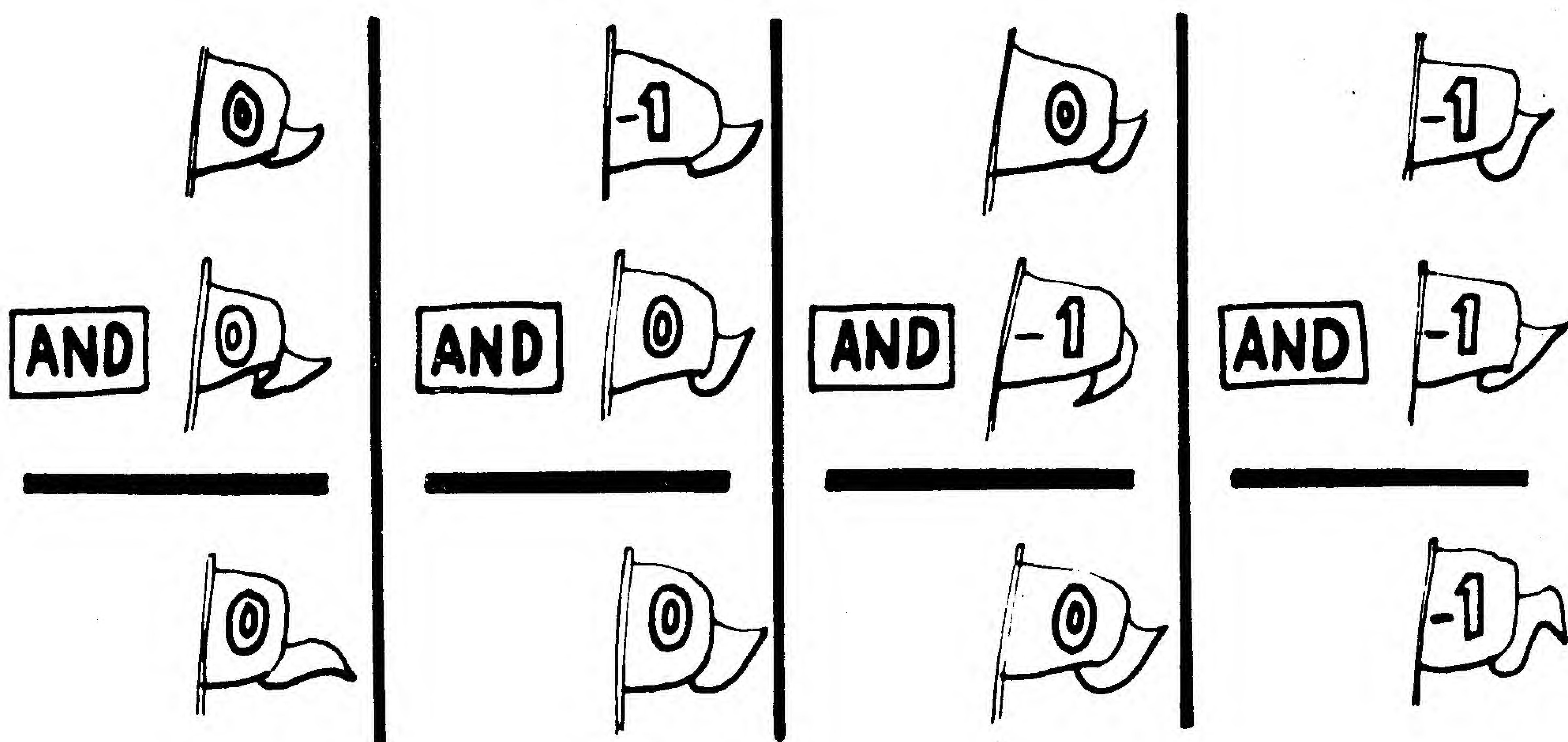
(Во многих Форт-системах для таких случаев предусмотрено слово **WITHING** (В ИНТЕРВАЛЕ). См. вопросы в конце главы.)

Другой комбинацией флагов является комбинация по принципу И. Здесь для получения истины оба флага должны быть истинны. Например, чтобы сделать сквозняк, нужно открыть обе двери: входную И выходную. Сравните с комбинацией ИЛИ: если входная ИЛИ выходная дверь (или сразу обе) будут открыты, то налетят мухи[1].

Форт включает слово **AND** (И). Ниже приводится таблица результатов операции **AND** над двумя флагами:

---

[1] *Для любознательных и начинающих.* Применение слов, аналогичных «или» и «и», при создании некоторых фрагментов прикладных программ называется *логикой.* Нотация для логических операций была разработана в XIX в. Дж. Булем. Этот раздел математики теперь называют булевой алгеброй. Таким образом, термин «булевский флаг» (или даже «булевская переменная») просто обозначает флаг, который будет использован в логической операции.



Иными словами, только пара значений «истина» дает в результате истину.

Допустим далее, что мы подбираем картонный ящик для дисковода, имеющего следующие параметры: высота — 6 дюймов, ширина — 19 дюймов, длина — 22 дюйма. Для того чтобы дисковод поместился в ящик, должны быть выдержаны все параметры: и высота, и ширина, и длина. Если параметры ящика заданы в стеке, то можно написать следующее определение:

```
: ОБЪЕМ  ( длина ширина высота -- )
   6 >  ROT 22 >  ROT 19 >  AND AND
   IF ."    Подходит "  THEN ;
```

Проверить слово ОБЪЕМ можно следующей фразой:

```
23 20 7 ОБЪЕМ Подходит ок
```

Третий вид комбинации флагов называется *исключающим ИЛИ*. В Форте эта операция выполняется словом **XOR**. Результат получается истинным только в тех случаях, когда *хотя бы один* из флагов истинен, но *не оба сразу*.

Ниже приводится пример применения такой операции. Слово ?ЗНАКИ выбирает из стека два числа. Если их знаки совпадают (оба числа положительны или оба отрицательны), то выдается сообщение «Знаки совпадают», в противном случае выдается сообщение «Знаки разные»:

```
: ?ЗНАКИ  ( n1 n2 -- )
   0<  SWAP 0<  XOR IF ." Знаки разные "
   ELSE  ." Знаки совпадают " THEN  ;
```

По мере усложнения ваших прикладных программ вы можете писать операторы Форта в виде текста на естественном языке в постфиксной форме, что очень легко читается. Только определите

где-нибудь отдельные слова, находящиеся внутри определения, которые будут проверять некоторое условие и оставлять в стеке флаг.

Следующий пример: вам предстоит выполнить фотосхемку, но прежде необходимо проверить освещенность И наличие пленки в фотоаппарате:

```
: ФОТОСъЕМКА   ?СВЕТ  ?ПЛЕНКА AND  IF СНИМОК THEN ;
```

Другой фрагмент, который может быть использован в программе обработки данных:

```
: СЛУЖБА-ЗНАКОМСТВ   С-ЮМОРОМ ОТЗЫВЧИВЫЙ AND
   ЛЮБИТ-ИСКУССТВО ЛЮБИТ-МУЗЫКУ OR  AND
     КУРИТ NOT  AND
     IF ." У нас имеется подходящая для вас кандидатура "  THEN ;
```

Здесь такие слова, как С-ЮМОРОМ и ОТЗЫВЧИВЫЙ, предназначены для выполнения проверки записи из дискового файла, содержащего информацию о возможных партнерах.

Обращаем ваше внимание на то, что до тех пор, пока мы не объясним некоторые тонкости использования слов **NOT**, **OR**, **AND** и **XOR**, их следует применять *только* с аргументами, являющимися логическими флагами, т. е. с нулем или отрицательной единицей. Все рассмотренные выше операции сравнения (за исключением операции « — », применяемой вместо «не равно») оставляют в стеке логические флаги. Поэкспериментируйте, чтобы посмотреть, как выполняются команды **NOT**, **OR**, **AND** и **XOR** с числами, отличными от 0 и — 1.

ДВА СЛОВА С ВСТРОЕННЫМИ ОПЕРАТОРАМИ IF

DUP

Слово **?DUP** дублирует вершину стека только в том случае, если там находится нулевое значение. Это помогает избавиться от лишних слов. Например, определение

```
: ?EMIT  ( c -- )  DUP IF EMIT ELSE DROP THEN ;
```

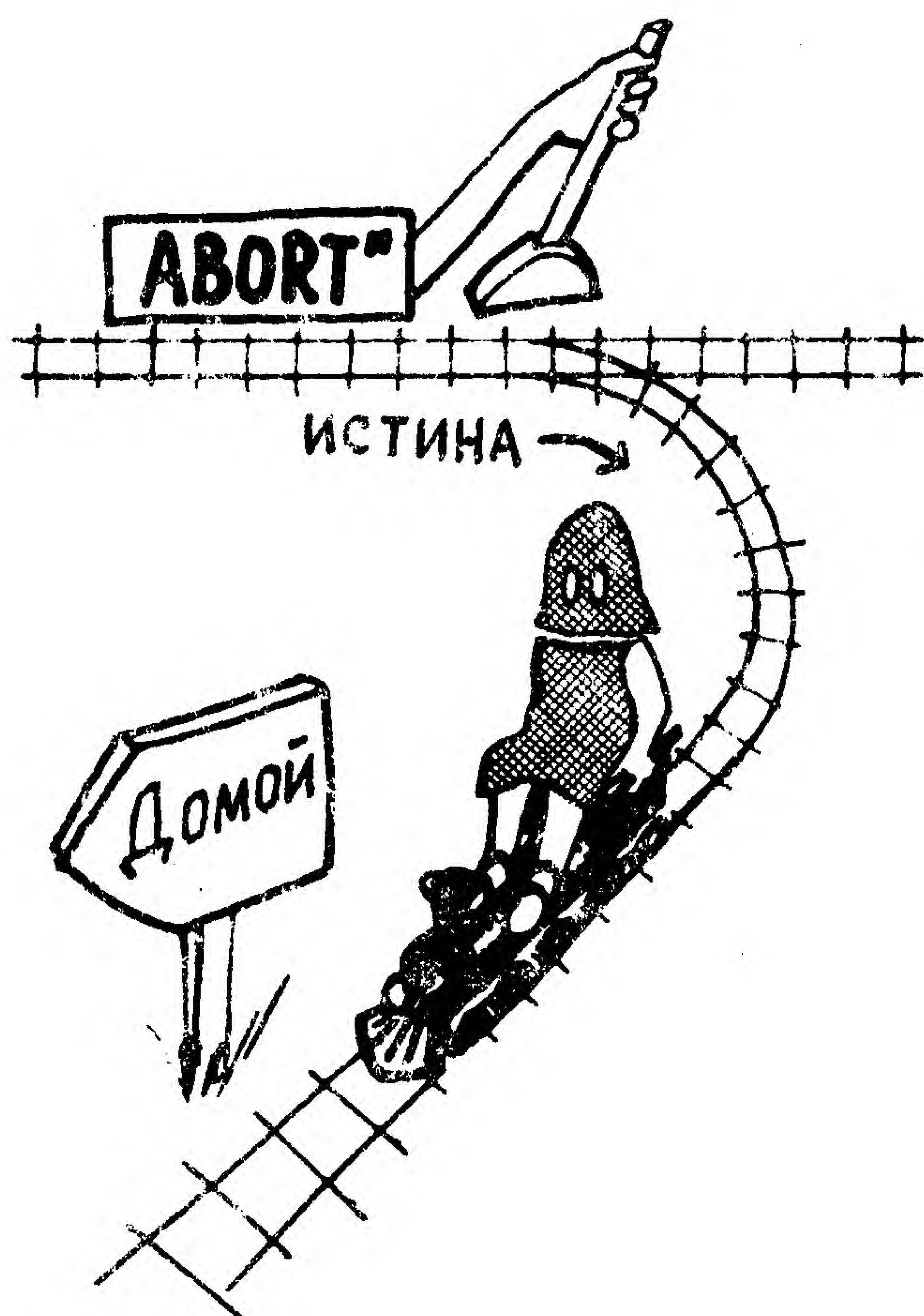
выдает на печать символ с любым кодом (кроме 0). Применяя **?DUP**, можно сократить наше определение:

```
: ?EMIT  ( c -- )  ?DUP IF EMIT THEN ;
```

ABORT"

В каком-то месте сложной прикладной программы может быть обнаружена ошибка (например, деление на нуль), которая проявляется в одном из слов низкого уровня. Когда это происходит, вы, естественно, хотите, чтобы компьютер прекратил вычисления и чтобы из стека были удалены все данные.

Если вы предполагаете, что подобная ошибка может произойти, можно воспользоваться оператором аварийного прекращения выполнения задачи **ABORT"**. Этот оператор проверяет значение флага в вершине стека и в случае его истинности прерывает вычисления. Оператор очищает стек и возвращает управление на терминал до поступления какого-либо сообщения. Оператор **ABORT"** также выводит имя последнего слова, обработанного текстовым интерпретатором, и предусмотренное вами для такой ситуации сообщение[1].



Проиллюстрируем изложенное примером:

```
: /ПРОВЕРКА  ( числитель знаменатель -- результат)
   DUP 0=  ABORT" Знаменатель нуль "  / ;
```

В этом определении, если знаменатель равен нулю, то любое число, оказавшееся в вершине стека, удаляется из последнего, а на терминал выводится сообщение:

```
8 0 /ПРОВЕРКА /ПРОВЕРКА Знаменатель нуль
```

Теперь в порядке эксперимента попытайтесь поместить слово /ПРОВЕРКА внутрь другого определения:

```
: ОБОЛОЧКА   /ПРОВЕРКА  ." Ответ равен "  . ;
```

[1] *Для профессионалов*. В Форте, кроме того, имеются слова: **QUIT** (ОКОНЧИТЬ), которое вызывает прекращение работы программы, но не очищает стек, и **ABORT** (ПРЕРВАТЬ), которое выполняет те же действия, что и **QUIT**, очищает стек, но не выводит сообщение. Мы рассмотрим эти слова в гл. 9.

и введите

```
8 4 ОБОЛОЧКА Ответ равен 2 ок
8 0 ОБОЛОЧКА ОБОЛОЧКА Знаменатель нуль
```

Обратите внимание на то, что когда слово /ПРОВЕРКА аварийно прерывает работу с помощью оператора **ABORT"**, оставшаяся часть ОБОЛОЧКА пропускается. Заметьте также, что выводится имя ОБОЛОЧКА, а не /ПРОВЕРКА.

Ниже приводится перечень слов Форта, рассмотренных в настоящей главе.

IF xxx ELSE yyy THEN zzz	IF: (? --)	Выполнение xxx, если ? истинно (ненулевое значение), и yyy, - если ? ложно. zzz выполняется независимо от выбранного варианта. Выражение yyy является необязательным.
=	(n1 n2 -- ?)	Занесение в стек истины, если n1 и n2 равны.
<>	(n1 n2 -- ?)	Занесение в стек истины, если n1 и n2 не равны.
<	(n1 n2 -- ?)	Занесение в стек истины, если n1 меньше n2.
>	(n1 n2 -- ?)	Занесение в стек истины, если n1 больше n2.
0=	(n -- ?)	Занесение в стек истины, если n является нулем (то есть истина меняется на ложь и наоборот).
0<	(n -- ?)	Занесение в стек истины, если n отрицательно.
0>	(n -- ?)	Занесение в стек истины, если n положительно.
NOT	(? ---?)	Изменение значения флага на противоположное.
AND	(n1 n2 -- И)	Доставление логического значения, согласно таблице операции AND .
OR	(n1 n2 -- ИЛИ)	Занесение в стек логического значения, согласно таблице операции OR .
XOR	(n1 n2 -- ИСКЛЮЧ-ИЛИ)	Занесение в стек логического значения, согласно таблице операции XOR .
?DUP	(n -- n n) или (0 -- 0)	Дублирование вершины стека только в том случае, если n является ненулевым значением.
ABORT" xxx"	(? --)	Если значение флага истинно, то вывод последнего проинтерпретированного слова и за ним заданного текста. Кроме этого очищаются стеки пользователя и возвращается управление на терминал. Если в стеке ложь, то не предпринимается никаких действий.

Обозначения:

n,n1 ... 16-разрядные числа со знаком. ? -- логическое значение (флаг)

ОСНОВНЫЕ ТЕРМИНЫ

ABORT. Часто встречающийся в информатике термин, обозначающий оператор, который моментально прекращает вычисления при возникновении аварийной ситуации. Его выполнение позволяет предотвратить получение бессмысленных результатов и, возможно, нарушение функционирования системы.

Ветвление. Изменение линейной последовательности вычислений в зависимости от условий, возникающих в процессе выполнения программы.

Вложенные структуры. Структуры ветвления, «вложенные» одна в другую.

Логика. Система представления условий в форме «логических переменных», которые могут быть либо истинными, либо ложными, и комбинаций этих переменных посредством таких логических операций, как «и», «или» и «не». Полученное в результате выражение может быть либо истинным, либо ложным.

Логическая операция AND. Два логических значения объединяются таким образом, что если *оба* они истинны, то и результат получается истинным, в противном случае результат ложный.

Логическая операция «ИСКЛЮЧАЮЩЕЕ-ИЛИ». Два логических значения объединяются таким образом, что если *хотя бы одно* из них (но не оба сразу) истинно, то результат является истинным, в противном случае результат ложный.

Логическая операция OR. Два логических значения объединяются таким образом, что если *хотя бы одно* из них истинно, то результат является истинным, в противном случае ложный.

Операция сравнения. Одно значение сравнивается с другим (например, определяется, не больше ли одно значение, чем другое) и устанавливается соответствующее значение флага. Последнее, как правило, проверяется условным оператором. В Форте при сравнении значение флага остается в стеке.

Условный оператор. Слово, такое, как IF, которое устанавливает последовательность вычислений в зависимости от некоторого условия (истина или ложь).

Флаг. Число, сигнализирующее о том, что некоторое известное условие является истинным или ложным.

УПРАЖНЕНИЯ

4.1. Какое значение оставляет в стеке выражение 0= NOT при следующих значениях аргумента: 1, 0, 200?

4.2. Объясните, как поведет себя артишок при указанных выше значениях.

4.3. Определите слово с именем РАЗРЕШЕНИЕ, которое берет из стека

значение, соответствующее возрасту некоторого лица, и выводит одно из двух сообщений (согласно установленным в вашей местности порядкам):

```
Употребление спиртных напитков разрешено
По возрасту не положено
```

4.4. Определите слово с именем ПРОВЕРКА-ЗНАКА, которое проверяло бы число, находящееся в стеке, и выдавало бы одно из трех сообщений: «ПОЛОЖИТЕЛЬНОЕ», «НУЛЬ» или «ОТРИЦАТЕЛЬНОЕ».

4.5. Напишите определение <> (не равно), которое в отличие от определения — (минус) всегда оставляло бы в стеке значение «истина» или «ложь».

4.6. Напишите определение **XOR**, используя другие логические операции, такие, как **AND**, **OR** и **NOT**.

4.7. В гл. 1 мы определили слово с именем STARS, но оно не выполняется правильно с аргументом, равным нулю. (В Форт-системах, соответствующих Стандарту-83, будет выведено 65 535 звездочек, в остальных — одна). Предложите на основе старого определения новый вариант STARS, который был бы свободен от указанного недостатка.

4.8. Напишите определение слова **NEGATE** (оно, между прочим, входит в состав Форт-систем), которое меняло бы знак числа в стеке на противоположный (4 при этом становится —4, и наоборот). После этого, используя **NEGATE**, создайте слово **ABS**, чтобы поместить в стек абсолютное значение заданного числа n (Слово **ABS** также есть в вашей системе.)

4.9. Напишите определение слова /UP, которое округляло бы остаток от деления, не равный нулю. (При решении задачи: «Сколько ящиков потребуется для упаковки n предметов?», для размещения округленного остатка потребуется дополнительный ящик.)

4.10. Напишите определение для слова WITHIN (В ИНТЕРВАЛЕ), которое выбирает из стека три аргумента

```
( n нижняя-граница верхняя-граница -- )
```

и помещает в стек истинное значение только в том случае, когда n находится внутри диапазона:

```
нижняя-граница ≤ n < верхняя-граница
```

4.11. Существует игра с угадыванием числа (программируя ее, вы получите больше удовольствия, чем в нее играя). Вначале вы вводите некоторое число в стек, причем можете сохранить это число в тайне, выполнив слово **PAGE**, которое очищает экран терминала. Затем вы просите своего партнера попытаться угадать задуманное число. Предполагаемое значение играющий должен ввести вместе со словом УГАДАЙ, например:

```
100 УГАДАЙ
```

Компьютер должен выдать ответ «Слишком большое», «Слишком маленькое» или «Правильно!». Напишите определение для слова УГАДАЙ в предположении, что после нескольких попыток ответа введенные числа будут находиться в стеке. После правильного ответа стек необходимо очистить.

4.12. С помощью операторов **IF ... ELSE ... THEN** напишите определение с именем ПРОПИСЬ, которое выводит число, находящееся в стеке, в виде слов на естественном языке. Диапазон выводимых чисел — от —4 до 4. Если в стеке находится число вне этого диапазона, то должно выдаваться сообщение «Выходит за диапазон». Например:

```
2 ПРОПИСЬ Два ок
-4 ПРОПИСЬ Минус четыре ок
7 ПРОПИСЬ Выходит за диапазон ок
```

Создайте по возможности короткое определение. (*Указание*: слово **ABS** выдает абсолютное значение числа, находящегося в стеке).

4.13. Используя созданное в упр. 4.10. определение WITHIN, разработайте еще одну игру с угадыванием чисел и назовите ее ЛОВУШКА. По условию игры вы вводите значение, не известное вашему партнеру, после чего он пытается угадать заданное число, вводя два числа, между которыми, по его мнению, заключено искомое значение, например:

```
0 1000 ЛОВУШКА Между ок
330 660 ЛОВУШКА Между ок
440 550 ЛОВУШКА Вне ок
330 440 ЛОВУШКА Между ок
```

и т. д. до тех пор, пока ваш партнер не отгадает ответ:

```
391 391 ЛОВУШКА Вы угадали! ок
```

Совет: вы можете предусмотреть в своей программе следующее: когда с искомым числом совпадает только одно число из задаваемого интервала, она не будет выдавать «Между».

Глава 5
ОПЕРАЦИИ НАД ЦЕЛЫМИ ЧИСЛАМИ

В этой главе мы введем несколько новых арифметических операций и в процессе изложения решим задачу по управлению положением десятичной точки, используя только целочисленную арифметику.

СОКРАЩЕННЫЕ ОПЕРАЦИИ

Рассмотрим сначала наиболее простые операции. Ниже приводятся слова, смысл которых вам очевиден[1]:

1+	(n -- n+1)	Добавление единицы.
1-	(n -- n-1)	Вычитание единицы.
2+	(n -- n+2)	Добавление двойки.
2-	(n -- n-2)	Вычитание двойки.
2*	(n -- n*2)	Умножение на два (арифметический сдвиг влево).
2/	(n -- n/2)	Деление на два (арифметический сдвиг вправо).

Существуют три причины, по которым желательно в вашем новом определении применять такие операции, как 1+ вместо 1 +. Во-первых, вы всякий раз экономите немного словарной памяти. Во-вторых, поскольку указанные слова определены в терминах машинного языка конкретного типа компьютера (чтобы учесть преимущества архитектуры данного компьютера), они выполняются быстрее, чем комбинация: 1 +. Наконец, вы экономите какое-то время на компиляции.

[1] *Для начинающих.* Понятие «арифметический сдвиг влево (вправо)» мы объясним позднее.

СМЕШАННЫЕ МАТЕМАТИЧЕСКИЕ ОПЕРАЦИИ

Приведем еще четыре математические операции. Как и в случае сокращенных операций, их функции должны быть вам понятны:

```
ABS      ( n -- :n: )      Помещение в стек абсолютной величины задан-
                           ного числа.
NEGATE   ( n -- -n )       Изменение знака на противоположный.


MIN      ( n1 n2 -- min)   Помещение в стек минимального из двух задан-
                           ных чисел.

MAX      ( n1 n2 -- max)   Помещение в стек максимального из двух за -
                           данных чисел.
```

Рассмотрим две простые задачи на использование слов **ABS** и **MIN**.

ABS.

Напишите определение для вычисления разности между двумя числами (независимо от порядка их следования):

```
: РАЗНОСТЬ  ( n1 n2 -- разность )  - ABS ;
```

Результаты получаются одинаковыми при любом порядке ввода:

```
52 37 РАЗНОСТЬ . 15 ok          или
37 52 РАЗНОСТЬ . 15 ok
```

MIN.

Напишите определение для вычисления суммы комиссионного сбора, которую получат продавцы мебели, если им обещано 50 дол. или $^1/_{10}$ часть от продажной цены при каждой сделке (в зависимости от того, какая из сумм окажется меньшей):

```
: КОМИССИОННЫЕ  ( цена -- комиссионные )  10 /  50 MIN ;
```

При трех различных значениях получается следующее:

```
600 КОМИССИОННЫЕ . 50 ok
450 КОМИССИОННЫЕ . 45 ok
50 КОМИССИОННЫЕ . 5 ok
```

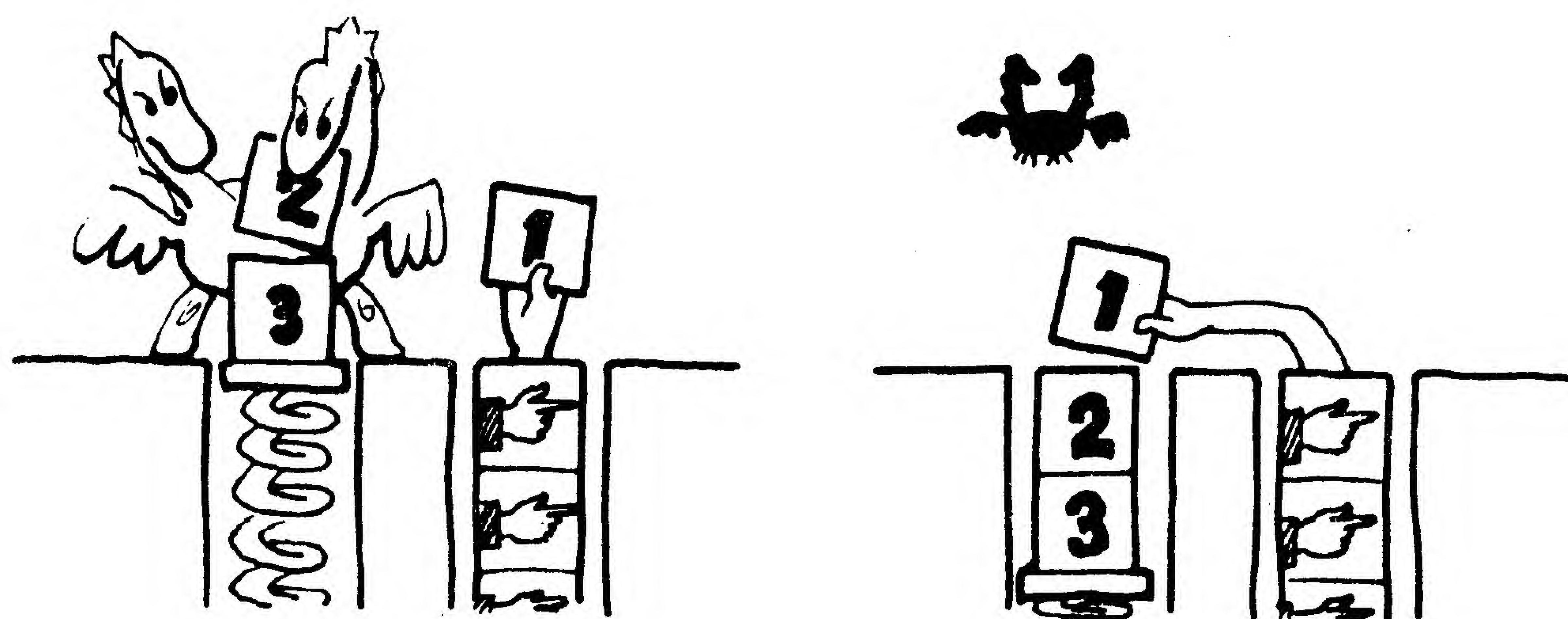
СТЕК ВОЗВРАТОВ

Ранее мы рассмотрели несколько операций со стеком. Однако существует еще целый ряд операций, с которыми вам предстоит познакомиться. До сих пор речь шла лишь о конкретном стеке так, как если бы он был один. Но на самом деле их два: стек данных и стек возвратов. Стек данных используется при программировании на Форте чаще, поэтому в тех случаях, когда это не вызывает недоразумений, он называется просто стеком.

Как вы уже видели, в стеке данных хранятся параметры (или аргументы), передаваемые от слова к слову. Стек же возвратов содержит любое число указателей, используемых Форт-системой для нахождения оптимального пути в лабиринте слов, которые вызывают выполнение *других* слов. Подробнее этот вопрос мы обсудим позднее.

Пользователь может рассматривать стек возвратов в качестве некоторого рода «дополнительной руки», которая «держит» значения в то время, когда вы работаете со стеком данных.





Подобно стеку данных, стек возвратов организован по принципу «последним внесен — первым выбран», поэтому в нем может храниться много значений. Но здесь имеется одна тонкость: данные, внесенные в стек возвратов, вы должны выбрать из него прежде, чем дойдете до конца соответствующего определения (точки

с запятой), так как в этот момент в вершине стека должен находиться некоторый указатель, используемый самой Форт-системой. Вы не имеете права обращаться к стеку возврата для передачи параметров от одного слова к другому.

Ниже дается перечень слов, связанных со стеком возвратов. Помните, что описание состояния стека относится здесь к *стеку данных*.

>R	(n --)	Выборка значения из стека данных и занесение его в стек возвратов.
R>	(-- n)	Выборка значения из стека возвратов и занесение его в стек данных.
R@	(-- n)	Копирование содержимого вершины стека возвратов без изменения его значения.

Слова >R и R> перемещают элемент стека данных в стек возвратов и элемент стека возвратов в стек данных соответственно. Показанные выше забавные рисунки иллюстрируют операции со стеком в состоянии (2 3 1 — 3 2 1), заданные выражением

```
>R SWAP R>
```

Каждое слово >R и соответствующее ему слово R >должны выполняться совместно в одном и том же определении или, если они выполняются в диалоговом режиме, в одной и той же вводимой строке (прежде чем вы нажмете клавишу RETURN).

Слово R @ только копирует значение из стека возвратов, но не удаляет его. Так что, введя выражение

```
>R SWAP R@
```

вы получите ожидаемый результат, но если при этом не уберете «мусор» до следующего двоеточия (или до того, как нажмете клавишу возврата каретки), то выведите систему из строя.

Поясним изложенное на примере. Допустим, вам настолько не повезло, что приходится вычислять значение полинома $ax^2 + bx+c$, причем задаваемые величины хранятся в стеке в следующем порядке:

```
( a b c x -- )
```

(Напоминаем, что операция возведения в степень должна выполняться первой.)

ОПЕРАЦИЯ	СТЕК ДАННЫХ	СТЕК ВОЗВРАТОВ
	a b c x	
>R	a b c	x
SWAP ROT	c b a	x
R@	c b a x	x
*	c b ax	x
+	c (ax+b)	x
R> *	c x(ax+b)	
+	x(ax+b)+c	

Попытаемся вычислить его. Загрузите следующее определение:

```
: ПОЛИНОМ   ( a b c x — n )
   >R SWAP ROT R@ *  + R> *  + ;
```

Теперь проверим его:

```
2 7 9 3 ПОЛИНОМ 48 ок
```

АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ НАД ЧИСЛАМИ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Вокруг Форта ведется много споров. Некоторые принципы, которых фанатически придерживаются программирующие на Форте, считаются неразумными в среде сторонников традиционных языков. Одним из предметов спора является вопрос выбора между представлениями числа «с фиксированной точкой» и «с плавающей точкой». Если вы уже поняли смысл этих терминов, то можете опустить данный материал и познакомиться с высказанной ниже нашей точкой зрения. Начинающим же полезно прочитать следующее объяснение.

Во-первых, что такое плавающая точка? Возьмем, к примеру, карманный калькулятор и посмотрим, как высвечивается результат на его индикаторе после ввода очередного значения:

ВВОД	НА ИНДИКАТОРЕ
1. 50 ×	1.5
2. 23	2.23
=	3.345

Десятичная точка «плавает» по индикатору по мере необходимости. Такой индикатор называется *индикатором с плавающей*

точкой. Представление с плавающей точкой — это способ записи чисел в компьютере в виде мантиссы и порядка. Например, 12 млн. можно записать как 12×10^6, поскольку 10^6 равно 1 млн. Во многих компьютерах 12 млн. должно быть записано в виде двух чисел 12 и 6, что воспринимается как 10^6, умноженное на 12. Число 3.345 будет записано так: 3345 и —3.

Идея представления чисел в форме с плавающей точкой состоит в том, чтобы компьютер мог представить необозримо большой диапазон чисел — от мизерных до астрономических — двумя сравнительно небольшими числами.

Представление с фиксированной точкой — другой способ записи чисел в память, при котором положение десятичной точки числа не запоминается. Например, при вводе суммы в долларах и центах все значения должны запоминаться в центах, а расположение десятичной точки должно «помнить» *не само число, а программа.*

Сравним для иллюстрации три представления чисел: общепринятое, с фиксированной точкой и с плавающей точкой:

ОБЩЕПРИНЯТОЕ ПРЕДСТАВЛЕНИЕ	С ФИКСИРОВАННОЙ ТОЧКОЙ	С ПЛАВАЮЩЕЙ ТОЧКОЙ
1.23	123	123(—2)
10.98	1098	1098(—2)
100.00	10000	1(2)
58.60	5860	586(—1)

Как видите, в представлении с фиксированной точкой все значения следуют одному шаблону. Компьютер в этом случае интерпретирует все числа как целые. Программа, прежде чем выдать ответ на экран терминала или алфавитно-цифровое печатающее устройство, вставляет десятичную точку после двух цифр справа.

ПОЧЕМУ ПРОГРАММИСТЫ ПРЕДПОЧИТАЮТ МАСШТАБИРОВАНИЕ

Многие опытные программисты, использующие традиционные языки программирования, воспринимают представление с плавающей точкой как нечто само собой разумеющееся. Их мнение можно выразить примерно так: «Почему я должен следить за перемещением десятичной точки? Для чего же тогда нужны компьютеры?». Вопрос поставлен правильно — он отражает основное преимущество реализации арифметических операций над числами с плавающей точкой. При переводе математических уравнений в машинный код такое представление чисел существенно облегчает жизнь программисту.

Однако многие прикладные Форт-программы должны работать в реальном масштабе времени. При этом компьютер используется для управления некоторым устройством или организации функционирования ряда дисплеев и клавиатур. Такие программы, для того чтобы «выжимать» из устройств максимальную производительность, нужно делать по возможности быстродействующими. Следовательно, программист зачастую заинтересован в максимизации производительности *аппаратных средств* в большей степени, чем в повышении эффективности *программирования*. Во многих случаях (например, при использовании карманных компьютеров) к тому же приходится экономить память.

Если в вашей программе некоторые вычисления должны повторяться миллионы раз, то требуемую скорость вы получите, выполняя арифметические операции над числами с фиксированной точкой. Действительно ли велик получаемый при этом выигрыш? Бесспорно. Время выполнения операций деления или умножения чисел с плавающей точкой намного превосходит время выполнения аналогичных операций над числами с фиксированной точкой. А при сложении или вычитании на подготовку и преобразование аргументов уходит столько же времени, сколько и на саму операцию. Большинство мини- и микрокомпьютеров «не думает» в терминах представлений с плавающей точкой и накладные расходы по организации выполнения соответствующих операций на них очень велики.

Свыше десяти лет программисты создают сложные прикладные Форт-программы на базе арифметики с фиксированной точкой. При этом им приходится решать самые изощренные задачи с использованием дифференциальных уравнений, быстрого преобразования Фурье, метода наименьших квадратов, линейной регрессии и т. д. То, что другие программисты делают на мощных универсальных ЭВМ, Форт-программисты выполняют на мини- и микрокомпьютерах, получая иногда общий выиграш в скорости вычислений.

Неверно, что Форт *не имеет возможности* поддерживать арифметику с плавающей точкой. Программистами созданы на Форте функции с плавающей точкой [1] — [7], а в ряде Форт-систем применяются сопроцессоры с плавающей точкой (отдельный чип, единственной функцией которого является реализация высокоскоростных операций над числами с плавающей точкой) [8], [9].

Большинство программистов считают, что операции над числами с плавающей точкой им необходимы, но это не так. При решении многих физических задач значения обрабатываемых и получаемых величин не выходят за пределы диапазона от единицы до нескольких тысяч и, следовательно, умещаются в 16-разрядное целое слово. (Некоторые вычисления могут потребовать 32-разрядного представления, что Форт в состоянии обеспечить.) К таким

задачам относится моделирование погоды, восстановление изображения, автоматическое измерение электрических цепей и т. п.

Масштабирование дает вам возможность работать с данными в 16-разрядном целочисленном диапазоне и поэтому отпадает надобность в расточительных операциях над числами с плавающей точкой. Не подумайте, что мы вынуждаем вас нарушать общепринятые правила и оставляем наедине со своими проблемами. Форт представляет в ваше распоряжение уникальный набор команд, так называемых *операций масштабирования*, которые поддерживают *аппроксимацию вещественных величин* и выполнение операций *с дробными значениями*. В следующем разделе мы познакомим вас с основными возможностями целочисленной арифметики.

ОПЕРАЦИЯ МАСШТАБИРОВАНИЯ */

Приведенная ниже арифметическая операция весьма полезна, хотя и выглядит несколько необычно:

```
*/        ( n1 n2 n3 --         Умножение, затем деление (n1*n2/n3).
                 результат )    Промежуточный результат 32-разрядный.
```

Например, допустим, что в стеке содержатся три числа:

```
( 225 32 100 -- )
```

*/ сначала перемножит 225 на 32, а затем разделит полученный результат на 100. Эта операция особенно удобна для решения таких задач, как вычисление процентов. В частности, вы можете определить слово % следующим образом:

```
: %  ( n % -- n')  100 */ ;
```

Введя число 225, а затем выражение

```
32 %
```

вы получите в стеке 32% от 225, т. е. 72[1].

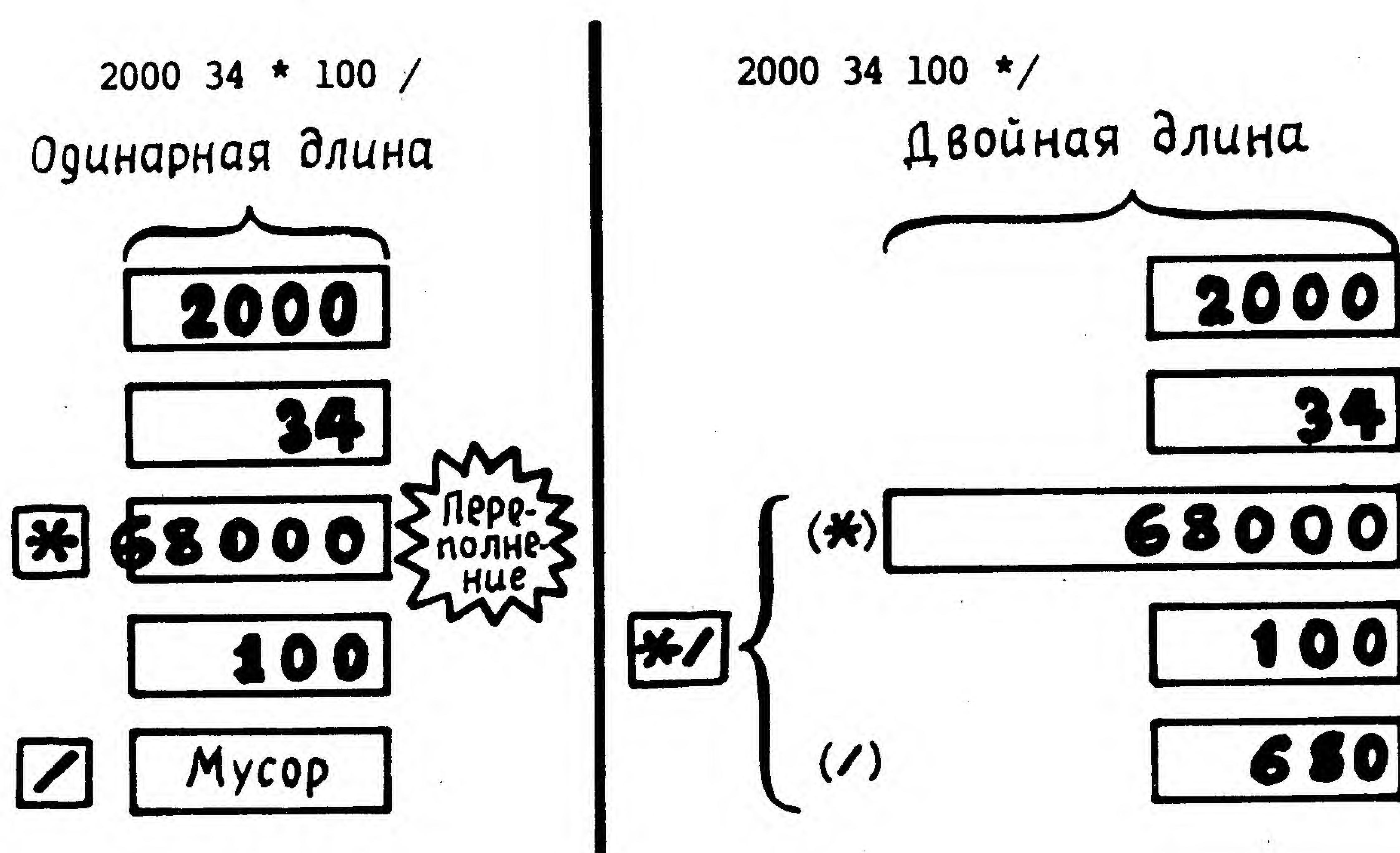
Команда */ это не простое сочетание команд * и /. Она использует для промежуточного результата число удвоенной длины. «Что это значит?», — спросите вы. Предположим, вам нужно вычислить 34% от 2000. Вспомним, что операции над числами одинар-

[1] *Для любознательных.* Способ, при котором сначала перемножаются два числа, а затем полученное произведение делится на 100, применяется и при выполнении подобных вычислений вручную.

ной точности, такие, как * и /, дают результаты в диапазоне от −32 768 до +32767. Если вы введете выражение

```
2000 34 * 100 /
```

(то получите неверный результат, поскольку промежуточное выражение (в нашем случае произведение) превышает 32767:



При выполнении же сдвоенной операции для промежуточного выражения применяется слово двойной длины, так что его разрядность позволяет помещать произведение любых двух чисел одинарной длины. Выражение

```
2000 34 100 */
```

дает правильный результат, так как конечное выражение находится в пределах диапазона одинарного представления чисел.

ОКРУГЛЕНИЕ

Из предыдущего примера вытекает другой вопрос: как производить округление? Решим такую задачу: сколько бананов нужно завезти в школьный буфет, если он рассчитан на 225 человек, и 32% из них обычно покупают бананы? Естественно, нас интересуют только целые бананы, так что в случае получения нецелого результата его нужно округлить. Сейчс слово % определено так, что все цифры справа от десятичной точки просто отбрасываются. Иными словами, результат «усекается»:

32% ОТ	РЕЗУЛЬТАТ
225 = 72.00	72 — абсолютно правильный
226 = 72.32	72 — правильный, округлен в меньшую сторону (усечен)
227 = 72.64	72 — усечен, но не округлен

Существует, однако, способ, при котором любое десятичное значение от 0,5 и выше, полученное в остатке, округляется до следующего целого. Для того чтобы найти «округленные проценты», мы должны определить слово R%:

```
: R%  ( n % -- результат )  10 */ 5 +  10 / ;
```

так что теперь выражение

```
227 32 R% .
```

даст в результате 73, т. е. правильно округленное до следующего целого числа.

Заметьте, что сначала мы делим на 10, а не на 100. При этом у нас появляется возможность добавить цифру 5 в позицию единиц числа, полученного в результате выполнения операции */.

ОПЕРАЦИЯ	СОДЕРЖИМОЕ СТЕКА
	227 32 10
*/	726
5 +	731
10 /	73

Окончательное деление на 10 приводит число к надлежащему виду, в чем вы можете убедиться самостоятельно[1].

Как недостаток такого способа округления необходимо отметить, что вы теряете одну значащую цифру в конечном результате, а именно вместо 32,767 мы можем получить только 3,276. Но если для вас это важно, вы всегда можете воспользоваться числами двойной длины (они будут рассмотрены позднее), которые также можно округлять.

ВОЗМОЖНОСТИ МАСШТАБИРОВАНИЯ

Решим простую задачу, допустим, что нужно вычислить $^2/_3$ от 171. По существу, есть два способа нахождения результата:

[1] *Для специалистов.* Существует и более быстрое определение:

```
: R%  ( n % -- результат )  50 */  1+ 2/ ;
```

1. Определить значение дроби $^2/_3$ путем деления числа 2 на число 3. При этом мы получим периодическую десятичную дробь, .666666 ... Далее можно умножить полученное значение на 171. Результат составит 113.9999999 и т. д., что не совсем точно, но он может быть округлен до 114.

2. Перемножить 171 и 2, а затем разделить полученное число 342 на 3, что дает в результате 114.

Заметим, что второй способ проще и намного точнее.

Большинство машинных языков поддерживает первый способ. В компьютере не могут храниться такие дроби, как $^2/_3$, на всякий случай. Их нужно выражать в виде десятичной дроби, например .666666 ...

Форт поддерживает второй способ. Операция */ позволяет вам получать дроби, аналогичные $^2/_3$, как в следующем примере:

```
171 2 3 */
```

Теперь, когда вы имеете некоторое представление о масштабировании, рассмотрим несколько более сложный пример. Допустим, требуется разделить 150 дол. в заданной пропорции[1]:

7.105	?
5.145	?
12.250	150

Эту задачу можно решить так:

(7.105/12.250) × 150 и
(5.145/12.250) × 150

Однако для обеспечения большей точности мы должны написать:

(7.105 × 150)/12.250 и
(5.145 × 150)/12.250

На Форте это будет выглядеть следующим образом:

```
7105 150 12250 */ . 87 ok
```

и

```
5145 150 12250 */ . 63 ok
```

Мы можем сказать, что значения 87 и 63 выбраны в «масштабе» к 7105 и 5145. Вычисление процентов, выполненное нами ранее, также представляет собой форму масштабирования. По этой причине операция */ называется *операцией масштабирования*.

Еще одной операцией масштабирования является операция */MOD:

```
*/MOD      ( n1 n2 n3 --
       n-остаток n-результат )
```

Умножение, затем деление (n1*n2/n3). Помещение на стек остатка и частного. Для промежуточного результата используется слово двойной длины.

Самостоятельно придумайте хороший пример на выполнение операции */MOD.

АППРОКСИМАЦИЯ ВЕЩЕСТВЕННЫХ ЧИСЕЛ

До сих пор мы применяли масштабирование для выполнения операции над вещественными числами. Масштабирование также может использоваться для представления иррациональных констант вещественными значениями, например числа π или $\sqrt{2}$. Фактически значение π представляется дробью 3.14159265358. Но для того чтобы оставаться в пределах арифметики с числами одинарной длины, мы должны записать эту дробь в виде выражения

```
31416 10000 */
```

что дает довольно хорошее приближение.

Теперь мы можем вычислить площадь круга по заданному радиусу. Переведем формулу πr^2 на язык Форт. Значение радиуса должно находиться в стеке, поэтому удвоим его с помощью DUP и умножим на самого себя, а затем получим конечный результат, применив операцию */:

```
: PI  ( n -- n')  31416 10000 */ ;
: ПЛОЩАДЬ  ( радиус -- площадь)  DUP *  PI ;
```

С помощью определения вычислим площадь круга с радиусом 10 дюймов:

```
10 ПЛОЩАДЬ . 314 ok
```

Для получения еще большей точности мы могли бы поискать другую пару чисел, которая давала бы лучшее приближение. Как ни странно, такая пара есть. Это дробь

```
355 113 */
```

обеспечивающая точность, бóльшую, чем шесть знаков после запятой, в то время как 31416 — обеспечивает менее четырех знаков.

Следовательно, наше новое, улучшенное определение будет иметь вид:

```
: PI  ( n -- n')  355 113 */ ;
```

Оказывается, вы можете хорошо аппроксимировать любую константу посредством множества различных пар целых чисел, значения которых меньше, чем 32768, с погрешностью менее 10−8[1].

ОПЕРАЦИИ НАД ДРОБНЫМИ ЧИСЛАМИ

Как вы видели, для масштабирования нецелые величины можно выразить в виде пары целых чисел. В некоторых программах вещественные числа могут понадобиться не только для масштабирования. Например, как сложить две дроби, не используя представление с плавающей точкой?

$$\frac{7}{34}+\frac{23}{99}=$$

На Форте вы можете сделать это с помощью команд так называемой *дробной арифметики*, реализующих операции над числами *с фиксированной точкой*.

При использовании дробной арифметики мы применяем масштабирование с подразумеваемым положением десятичной точки. Но вместо масштабирования посредством умножения на степень 10 (как принято при вычислениях вручную) мы будем масштабировать путем умножения на степень 2 (как принято при вычислении на компьютере). Таким образом, выражение «десятичная точка» уместно заменить выражением «двоичная точка»[2].

---

[1] *Для истинных математиков.* Ниже приводится небольшая таблица вещественной аппроксимации различных констант:

Значения	Аппроксимация	Погрешность
π = 3.141 ...	355/ 113	8.5×10^{-8}
$\sqrt{2}$ = 1.414 ...	19601/13860	1.5×10^{-9}
$\sqrt{3}$ = 1.732 ...	18817/10864	1.1×10^{-9}
e = 2.718 ...	28667/10546	5.5×10^{-9}
$\sqrt{10}$ = 3.162 ...	22936/ 7253	5.7×10^{-9}
$\sqrt[12]{2}$ = 1.059 ...	26797/25293	1.0×10^{-9}
$\log_{10}$ 2/1.6384 = 0.183 ...	2040/11103	1.1×10^{-8}
ln2/16.384 = 0.042 ...	485/11464	1.0×10^{-7}
.001° /22-bit rev = 0.858 ...	18118/21109	1.4×10^{-9}
arc-sec/22-bit rev = 0.309 ...	9118/29509	1.0×10^{-9}
c = 2.9979248	24559/ 8192	1.6×10^{-9}

[2] *Для начинающих.* Этот вопрос подробнее рассматривается в гл. 7.

Допустим, вы ввели следующее определение:

```
: +1  ( -- масштабная-единица )  16384
```

Здесь выбран масштаб, при котором число 16384 представляет положительную единицу (константы будут введены в гл. 8). В двоичной системе счисления число 16384 выглядит следующим образом:
0100000000000000
т. е. это единица в соответствующем масштабе с подразумеваемой двоичной точкой.

Теперь добавьте к Форту две новые арифметические операции: дробное умножение и дробное деление, выполняемые с учетом выбранного нами масштаба.

```
: *.  ( n n -- n)  +1  */ ;
: /.  ( n n -- n)  +1  SWAP */ ;
```

Что же у вас получилось? При делении единицы на единицу должна получиться единица.

```
1 1 /. . 16384 ok
```

(Напомним, что в выбранном масштабе 16384 это единица.) Теперь разделите 1 на 2:

```
1 2 /. . 8192 ok
```

Здесь 8192 представляет половину единицы (половина от 16384). Следовательно, вы можете решить поставленную задачу таким образом:

```
7 34 /.  23 99 /. +
```

Обратите внимание на последнюю операцию. Для сложения двух дробей мы применяем хорошо знакомый вам знак +. Конечно, пока это не совсем удобно, так как вы еще не умеете получать ответы в прежней форме. Для того чтобы представить результат снова в десятичной системе, необходимо ввести

```
10000 *. . 4381 ok
```

Ответ составит 4381/10000, или в более привычном виде 0.4381.

С помощью дробной арифметики, используя быстрые операции над целыми числами, вы можете складывать, вычитать, умножать

и делить даже вещественные числа. В вычислительных задачах (в которых выполняется множество арифметических действий), скорость вычислений считается самым важным параметром. Перевод же чисел в удобочитаемый вид (в десятичную систему) не столь существен, поскольку выводить нужно лишь конечный результат. Более того, в таких приложениях, как графика и робототехника, *вообще не требуется* перевод результата в десятичную систему — получаемая информация должна быть понятна графическому устройству, плоттеру, руке робота или чему-то еще.

Имеет смысл решать на компьютере задачи теми средствами, к которым он приспособлен, а не навязывать ему образ действий, являющийся следствием нашего школьного образования.

Нам все-таки хотелось бы вводить и выводить дроби с помощью привычных обозначений. Вы были бы не против, если бы Форт-система выводила результат с десятичной точкой в соответствующем месте? А для этого нужно всего лишь одно слово, представляющее число в традиционном формате. Воспользуемся средствами, описанными в гл. 7[1]:

```
: #.####   DUP  ABS  0 <#  # # # #  46 HOLD  # ROT SIGN  #>
   TYPE SPACE ;
: .F   ( дробь -- )   10000 *.   #.#### ;
```

Теперь наше выражение будет иметь вид

```
7 34 /.  23 99 /.  + .F 0.4381 ok
```

Это не представление с плавающей точкой, но результат в принципе такой же и получен гораздо быстрее.

Допустим, вы хотите, чтобы *вводимые* аргументы выглядели как вещественные числа, например:

.1250 + .3750 = ?

Для того чтобы выполнить такую операцию на Форте, в первую очередь вам потребуется слово, преобразующее числа с десятичной точкой в масштабированную дробь. Детали мы объясним вам в гл. 7, а для своих текущих потребностей вы можете определить, например, слово

```
: D>F  ( d -- дробь)  DROP  10000 /. ;
```

(D > F означает перевод из числа двойной длины в дробное. DOUBLE — двойной, FRACTION — дробь). Теперь можно вводить следующие выражения:

```
.1250 D>F .3750 D>F + .F 0.5000 ok
```

---

[1] *Для пользователей фиг-Форта, систем полиФорт, созданных до введения Стандарта-83 и других старых систем.* Перед **SIGN** слово **ROT** должно быть опущено.

Обратите внимание на то, что необходимо дополнять исходные данные до четырех десятичных знаков.

Вы можете выполнить умножение двух дробей посредством *.

```
.7500 D>F  .5000 D>F  *.  .F .3750 ok
```

Интересно отметить, что если вы выполняете операцию умножения *. дроби на целое, то результат будет целым, например:

```
28  .5000 D>F  *.  . 14 ok
```

Применяя операцию /., можно разделить, скажем, —0.3 на 0.95:

```
-.3000 D>F  .9500 D>F /.   .F -0.3160 ok
```

Выполняя ту же операцию над двумя целыми, вы получите дробный результат:

```
22 44 /. .F .5000 ok
```

Если же вы делите посредством /. целое на дробь, то в результате получите целое. Обозначив символом f дробь, а символом i целое, мы можем построить следующую таблицу выполнения операций:

ОПЕРАЦИЯ	РЕЗУЛЬТАТ
f f +	f
f f —	f
f i *	f
i f *	f
f i /	f
f f *.	f
f i *.	i
i f *.	i
f f /.	f
i i /.	i
i f /.	i

Использование двоично-ориентированного масштаба, например числа 16384, а не десятично-ориентированного, например 10000, позволяет обеспечить в пределах 16 разрядов бóльшую точ-

ность (точность повышается в отношении 16 к 10). При этом операции */ и /. кодируются на ассемблере чрезвычайно эффективно. Но и число 16384 мы выбрали в качестве единицы произвольно. Если бы было выбрано число 256, то получилось бы восемь разрядов (включая знак) слева от двоичной точки и восемь разрядов справа. Этот метод при необходимости можно применить и к 32-разрядным числам.

Операции *. и /. можно выполнять над тригонометрическими функциями, поскольку значение угла представляется как функция длины окружности, выраженная дробью (в интервале от 0 до 1). Перевод радиан в градусы и обратное преобразование осуществляются весьма просто.

ЗАКЛЮЧЕНИЕ

Итак, вы познакомились с масштабированием, методами округления и аппроксимации вещественных чисел и операциями над числами с фиксированной точкой. Для того чтобы вы чувствовали себя увереннее при решении сложных математических задач, где необходимо следить за правильностью выбора масштаба в процессе решения посмотрите второй пример в гл. 12.

Как уже упоминалось, вам ничто не мешает дополнительно ввести в Форт операции над числами с плавающей точкой. Но такие средства не очень подходят Форту, поскольку его достоинства — это компактность, высокая скорость выполнения программ, простота и элегантность. Он как бы «отторгает» от себя все то, что не является насущной необходимостью. Разумно используя масштабирование и числа двойной длины, вы избавитесь от дорогостоящих операций над числами с плавающей точкой.

Операции над числами с плавающей точкой имеет смысл применять в следующих случаях:

- если вы хотите использовать ваш компьютер как калькулятор для одноразовых вычислений;
- когда время программирования представляется более существенным фактором, нежели время выполнения вычислений в вашей программе;
- для обработки данных в большом динамическом диапазоне (превышающем диапазон от −2 биллионов до +2 биллионов).

Все перечисленные доводы являются серьезными. Однако существует целый ряд систем, где вы не должны платить за возможность выполнения операций над числами с плавающей точкой.

Ниже приводится список слов Форта, используемых в данной главе:

1+	(n -- n+1)	Добавление единицы.
1-	(n -- n-1)	Вычитание единицы.
2+	(n -- n+2)	Добавление двойки.
2-	(n -- n-2)	Вычитание двойки.
2*	(n -- n*2)	Умножение на два (арифметический сдвиг влево).
2/	(n -- n/2)	Деление на два (арифметический сдвиг вправо).
ABS	(n -- :n:)	Помещение в стек абсолютной величины заданного числа.
NEGATE	(n -- -n)	Изменение знака на противоположный.
MIN	(n1 n2 -- min)	Помещение в стек минимального из двух заданных чисел.
MAX	(n1 n2 -- max)	Помещение в стек максимального из двух значений
>R	(n --)	Выборка значения из стека данных и занесение его в стек возвратов.
R>	(-- n)	Выборка значения из стека возвратов и занесение его в стек данных.
R@	(-- n)	Копирование содержимого вершины стека возвратов без изменения его значения.
*/	(n1 n2 n3 -- результат)	Умножение, затем деление (n1*n2/n3). Промежуточный результат 32-разрядный.
*/MOD	(n1 n2 n3 -- n-остаток n-результат)	Умножение, затем деление (n1*n2/n3). Помещение на стек остатка и частного. Для промежуточного результата используется слово двойной длины.

ОСНОВНЫЕ ТЕРМИНЫ

Арифметические операции над числами с фиксированной точкой. Операции над числами, в которых не указано положение десятичной точки. Вместо этого для некоторой группы чисел программа подразумевает десятичную точку в какой-то позиции или хранит позицию десятичной точки для этой группы чисел в виде отдельного числа.

Арифметические операции над числами с плавающей точкой. Операции над числами, в которых положение десятичной точки указано. Программа должна интерпретировать истинное значение каждого отдельного числа, прежде чем какая-либо арифметическая операция может быть выполнена.

Дробная арифметика. Арифметика, где можно выражать дроби с помощью целых чисел, а за положением точки следит сам программист.

Масштабирование. Процесс умножения (или деления) какого-то числа в определенной пропорции. То же относится к процессу

умножения (или деления) числа на степень десяти так, чтобы все значения в некотором наборе данных могли бы быть представлены как целые в предположении, что десятичная точка находится в одной и той же позиции для всех чисел, или на степень 2 в случае дробной арифметики.

Промежуточный результат двойной точности (длины). Значение двойной длины временно создается двухшаговой операцией, такой, как */, тогда, когда промежуточное значение (результат первой операции) может превышать число одинарной длины, даже в том случае, если исходные аргументы и конечный результат не выходят за пределы одинарной точности.

Стек данных. В Форте — это участок памяти, который служит общей областью для передачи аргументов между различными операциями (чисел, флагов и т. д.).

Стек возвратов. В Форте — это участок памяти, который Форт-система использует для хранения «адресов возврата» в отличие от других объектов (будет рассмотрен в гл. 9). Пользователь имеет право хранить значения в стеке возвратов временно, при определенных условиях.

УПРАЖНЕНИЯ

5.1. Укажите различия между —1 и 1—.

5.2. Переведите следующее алгебраическое выражение в форму определения Форта:

$$-\frac{ab}{c}$$

при состоянии стека ($a\ b\ c$ —).

5.3. При состоянии стека (6 70 123 45 —) напишите выражение, которое бы инициировало печать наибольшего из этих значений.

5.4. а) Определите слово 2ПОРЯДОК, которое из имеющихся в стеке двух чисел располагало бы в вершине большее, а оставшееся — под ним.

б) Определите слово 3ПОРЯДОК, которое располагало бы три заданных числа в стеке так, чтобы большее было в его вершине.

в) Вспомните определение ОБЪЕМ из гл. 4. Перепишите его, используя определение 3ПОРЯДОК, так, чтобы пользователь мог вводить измерения в любом порядке.

Практикум в масштабировании

5.5. *Гистограмма* — это графическое представление серии значений, каждая из которых выражена высотой или длиной некоторого отрезка. Определите слово с именем РИСУЙ — компонент вашей программы по созданию гистограмм. По заданному значению в диапазоне от 0 до 100 слово РИСУЙ должно вывести на экран горизонтальную линию из звездочек, графически представляющую это заданное значение.

Трудность заключается в том, что на экране только 80 колонок. Таким образом, значение 100 должно соответствовать 80 звездочкам, значение 50 — 40 звездочкам, значение 0 — 0 звездочкам и т. д. (Начинайте ваше определение с команды **CR** и используйте вариант слова STARS из упр. 4.7.)

5.6. В режиме калькулятора переведите указанные значения температур из одной шкалы в другую по формулам

$$°C = \frac{°F - 32}{1,8};$$

$$°F = (°C \times 1,8) + 32; \quad °K = °C + 273.$$

Выразите все аргументы и результаты целыми числами (в градусах):
а) 0°F в °C;
б) 212° F в °C;
в) −32° F в °C;
г) 16° C в °F;
д) 233° K в °C.

5.7. Определите слова для выполнения преобразований из упр. 5.3. Используйте следующие имена:

```
F>C   F>K   C>F   C>K   K>F   K>C
```

Проверьте их выполнение с приведенными выше значениями.

ЛИТЕРАТУРА

1. Bowhill, Sidney A., "A Variable-Precision Floating-Point System for Forth," *1983 FORML Conference Proceedings*, Asilomar, California.
2. Bumgarner, John O., and Jonathan R. Sand, "Dysan IEEE P-754 Binary Floating Point Architecture," *1983 Rochester Forth Conference Proceedings*, pp. 185–94.
3. Jesch, Michael, "Floating Point in FORTH?" *1981 FORML Conference Proceedings*, pp. 61–78; reprinted as "Floating Point FORTH?", *Forth Dimensions*, 4/1, May–June 1982, pp. 23–25.
4. Harwood, James V., "FORTH Floating Point," *1981 Rochester Forth Standards Conference Proceedings*, p. 189.
5. Monroe, Alfred J., "Forth Floating-Point Package," *Dr. Dobb's Journal*, 7/9, September 1982, pp. 16–29.
6. Petersen, Joel V., and Michael Lennon, "NIC-FORTH and Floating Point Arithmetic," *1981 Rochester Forth Standards Conference Proceedings*, pp. 213–17.
7. Duncan, Ray, and Martin Tracy, "The FVG Standard Floating-Point Extension," *Dr. Dobb's Journal*, 9/9, September 1984, pp. 110–15.
8. Redington, Dana, "Forth and Numeric Co-processors: An Extensible Way to Floating-Point Computation," *Conference Proceedings of the Eighth WCCF*, 3, pp. 368–73, March 18–20, 1983.
9. Redington, Dana, "Stack-Oriented Co-Processors and Forth," *Forth Dimensions*, 5/3 September–October 1983, pp. 20–22.

Глава 6
ЦИКЛИЧЕСКИЕ СТРУКТУРЫ

В гл. 4 мы обсуждали вопрос о том, как нужно составлять программы, чтобы компьютер «принимал решения», выбирая одну из альтернатив определения в зависимости от результата проверки конкретного условия. Условный переход является одним из тех достоинств, которые делают компьютер незаменимым.

Теперь вы должны научиться писать определения, позволяющие повторять тот или иной фрагмент несколько раз в зависимости от заданных условий. Такой вид структуры управления называется *циклом*. Способность выполнять циклы — это второе и, вероятно, наиболее важное достоинство компьютеров. Если вы можете составить программу проверки одной платежной ведомости, то вы можете составить программу проверки и тысячи подобных ведомостей.

В настоящей главе мы рассмотрим циклы, в которых выполняются простые действия, например вывод чисел на вашем терминале.

ЦИКЛЫ СО СЧЕТЧИКОМ

Одна из циклических структур называется *циклом со счетчиком*. Здесь вы сами определяете число повторений цикла. На Форте для этого нужно задать начальное и конечное числа перед словом **DO** (в обратном порядке), а затем поместить слова, выполнение которых вы хотите повторять, между словами **DO** (ВЫПОЛНИТЬ) и **LOOP** (ЦИКЛ). Например:

```
: TECT   10 0 DO   CR   ." Привет "   LOOP ;
```

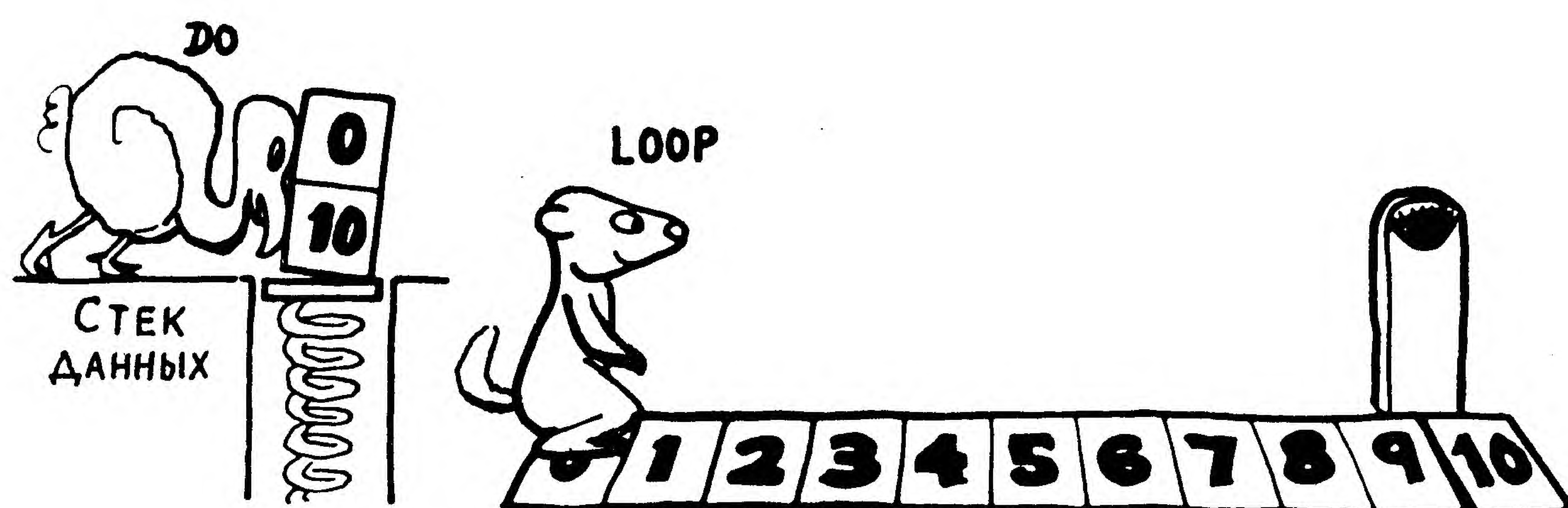
будет осуществлять возврат каретки и выводить слово ПРИВЕТ десять раз, потому что, если вычесть нуль из 10, вы получите 10.

```
ТЕСТ
ПРИВЕТ
ПРИВЕТ
ПРИВЕТ
ПРИВЕТ
ПРИВЕТ
ПРИВЕТ
ПРИВЕТ
ПРИВЕТ
ПРИВЕТ
ПРИВЕТ
```

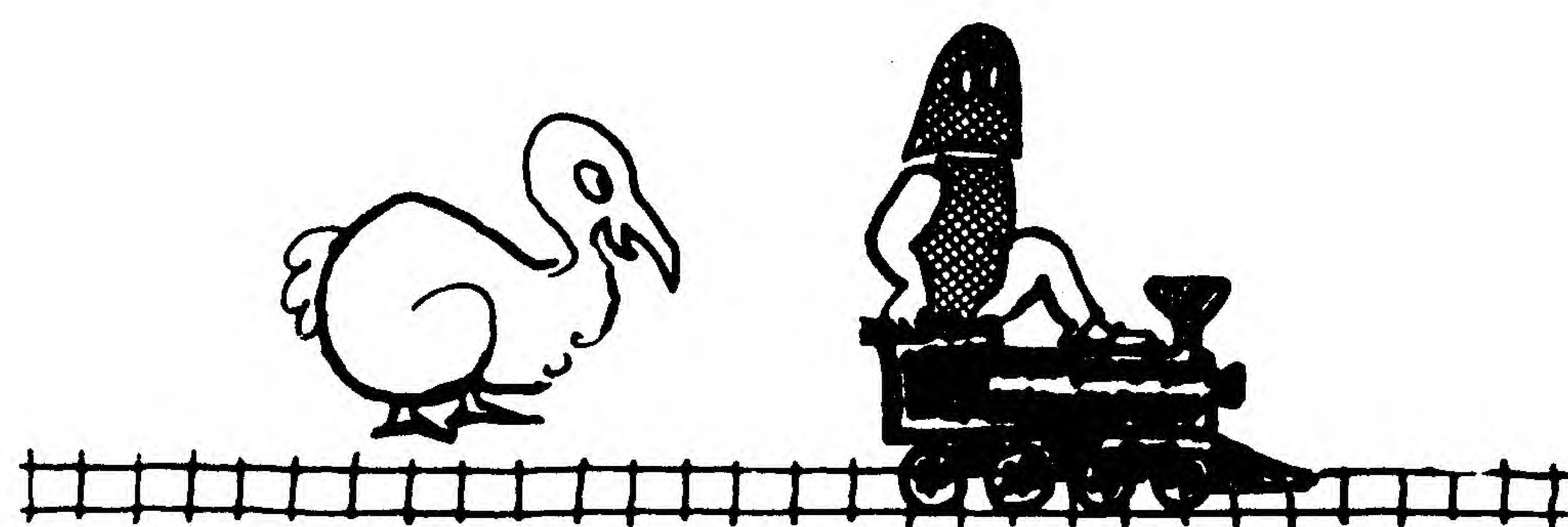
Как и оператор **IF ... THEN**, относящийся к операторам управления, оператор **DO ... LOOP** должен размещаться в пределах (одного) определения. Число 10 в приведенном примере называется *границей цикла* (limit), а нуль — ***индексом*** (index). Общий вид оператора цикла со счетчиком:

граница индекс DO ... LOOP

Теперь посмотрим, что происходит внутри цикла **DO ... LOOP**:



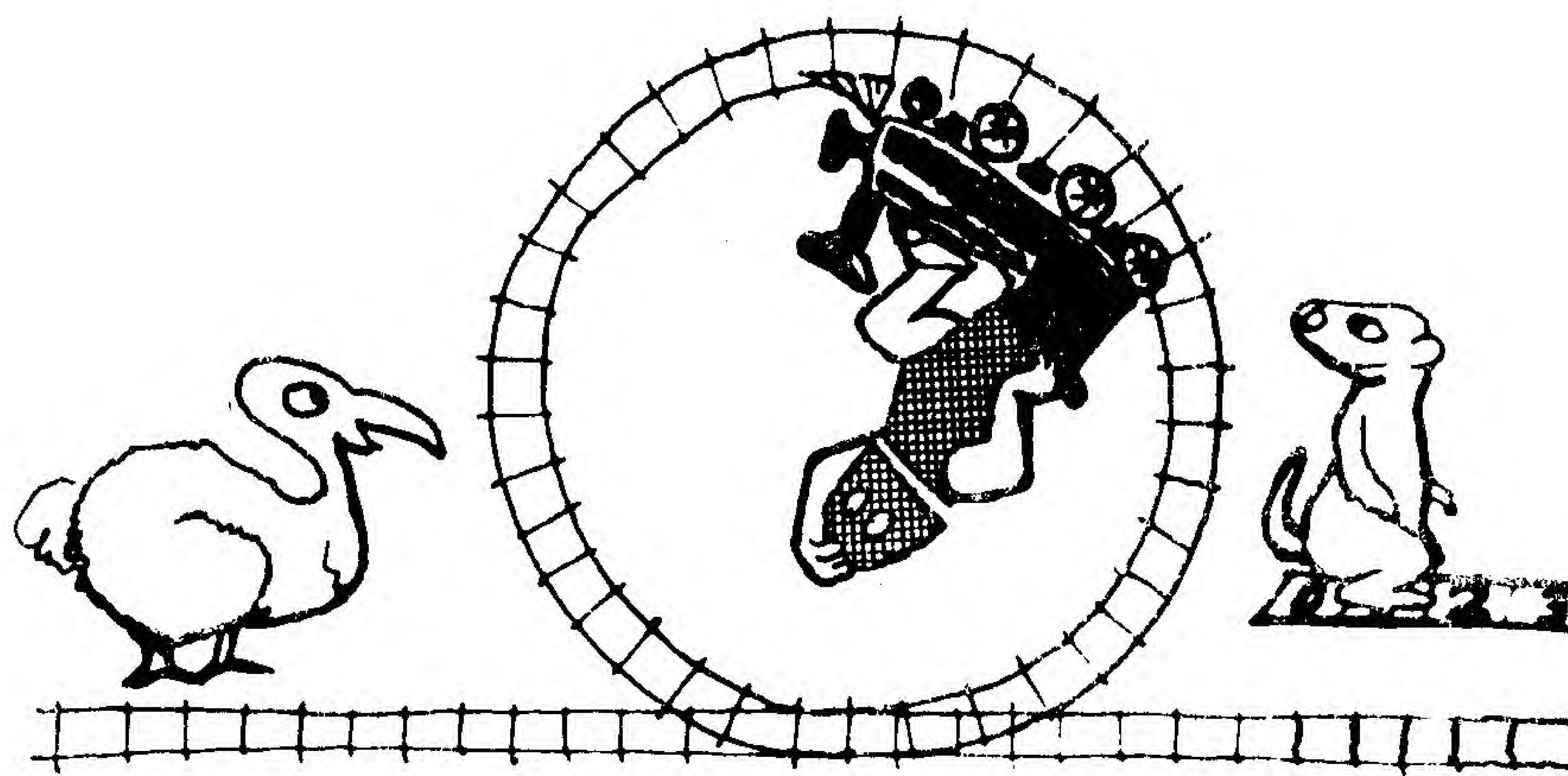
Слово **DO**[1] берет из стека данных **границу и индекс и** подготавливает их для слова **LOOP**, которое по **этим данным будет** управлять циклическим процессом.



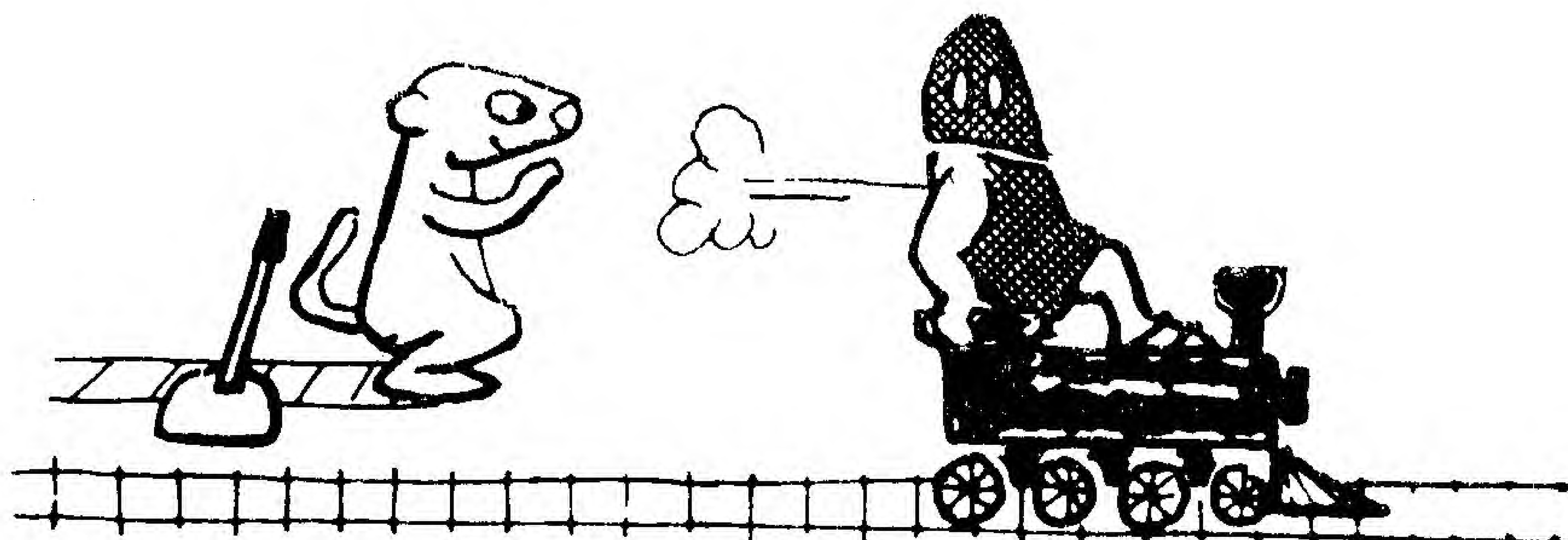
Далее выполняются слова, находящиеся внутри цикла.

[1] Единоутробный братец птички ДОДО.









Получив управление, **LOOP** продвигается на единицу, после чего управление возвращается к **DO**. Когда **LOOP** «пересекает» финишную черту, электронный глаз переключает стрелку, разрешая тем самым выполняться операторам, следующим за словом **LOOP**.

Слово **I** помещает в стек данных текущее значение *индекса* (значение, на котором в этот момент стоит **LOOP**) на каждом шаге циклического процесса. Рассмотрим определение

```
: ДЕКАДА   10 0 DO  I .  LOOP ;
```

которое выполняется так, как показано ниже:

```
ДЕКАДА 0 1 2 3 4 5 6 7 8 9 ok
```

Заметьте, что цикл выполняется всего 10 раз — от 0 до 9. Мы не добираемся до 10, потому что **LOOP** пересекает границу в тот момент, когда 9 сменяется числом 10.

Вы можете выбрать любые числа, но так, чтобы значение границы превышало значение индекса:

```
: ПРИМЕР    -243 -250  DO  I .  LOOP ;

ПРИМЕР -250 -249 -248 -247 -246 -245 -244 ok
```

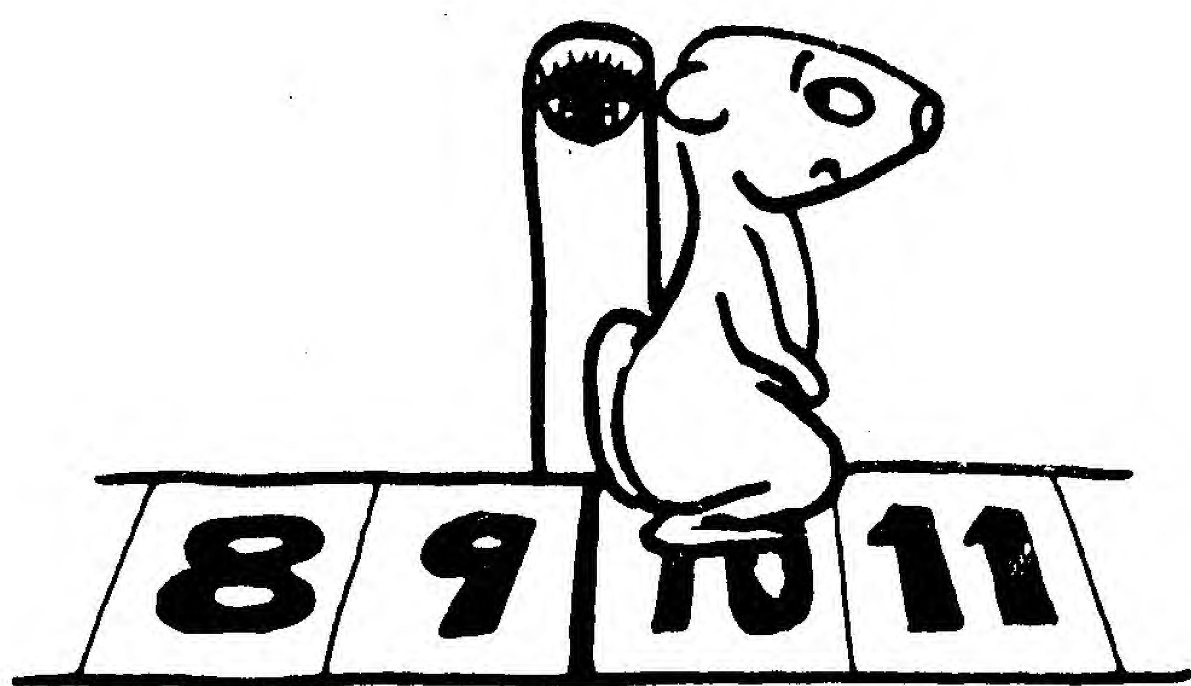
И в этом примере значение границы (—243) больше, чем начальный индекс (—250).

ОГРАНИЧЕНИЯ НА ВЫПОЛНЕНИЕ ЦИКЛА

Есть несколько важных правил, которых нужно придерживаться, программируя циклы **DO**. Прежде всего в операторе **DO** начальный индекс никогда не должен совпадать с границей. Если вы хотите, чтобы цикл не был выполнен ни разу, и определили его следующим образом:

```
: ОШИБКА   10 10 DO  I .  LOOP ;
```

то не получите ожидаемого результата. Результат будет таков:



К сожалению, **LOOP** уже переступил финишную черту и не попадет под электронный глаз в течение длительного времени (до тех пор, пока не вернется назад через 65 535 шагов)[1].

Если вы не уверены в том, что граница и индекс в каком-то вашем определении не совпадут, используйте вместо **DO** слово **?DO**, которое, как и первое, берет из стека (границу индекс --), но сразу передает управление **LOOP** в том случае, когда граница и индекс совпадают.

Возвращаясь к слову STARS из гл. 1, напишем его правильное определение:

```
: STARS  ( число-звездочек)   0 ?DO  42 EMIT  LOOP ;
```

При отсутствии в вашей системе слова **?DO** используйте следующее определение:

```
: STARS  ( число-звездочек)   ?DUP IF  0 DO  42 EMIT  LOOP  THEN ;
```

С какими неприятностями вы еще можете столкнуться? В большинстве Форт-систем при выполнении цикла **DO** текущие значения счетчика хранятся в стеке возвратов (введенном в гл. 5). Это при-

---

[1] *Для пользователей систем, созданных до принятия Стандарта-83.* В более старых системах цикл выполнился бы один раз (вместо 64К раз), но и это не то, что вам нужно.

водит к некоторым ограничениям. В частности, слово **I** можно употреблять только в том определении, в котором используются слова **DO** и **LOOP**. Вы не имеете права ввести такой текст:

```
: ТЕСТ   I . ;
: ДЕКАДА   10 0 DO  ТЕСТ  LOOP ;
```

Если вспомогательное определение (ТЕСТ) требует значения индекса, вы должны передать его через стек, например:

```
: ТЕСТ  ( индекс -- ) . ;
: ДЕКАДА   10 0 DO  I ТЕСТ  LOOP ;
```

Отметим еще несколько ограничений. Вы не должны перед словом **DO** помещать в стек возвратов с помощью слова **>R** промежуточные значения с тем, чтобы использовать их внутри цикла. Если же вы запрограммировали внесение некоторого значения в стек возвратов внутри цикла, то обязаны перед выходом из цикла, до начала выполнения **LOOP** (или слова **LEAVE**, которое мы рассмотрим ниже), запрограммировать его удаление словом **R>**.

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ОПЕРАТОРА ЦИКЛА DO

Вы можете оставить в стеке какое-нибудь число, которое будет служить аргументом для некоторого объекта внутри оператора цикла. Например,

```
: ПРОИЗВЕДЕНИЯ  ( n)  CR  11 1 DO  DUP I * . LOOP  DROP ;
```

даст вам следующий результат:

```
7 ПРОИЗВЕДЕНИЯ
7 14 21 28 35 42 49 56 63 70 ok
```

или

```
8 ПРОИЗВЕДЕНИЯ
8 16 24 32 40 48 56 64 72 80 ok
```

Здесь вы просто умножаете текущее значение индекса на n на каждом шаге цикла. Заметьте, что необходимо размножить число n с помощью **DUP** внутри цикла, чтобы его копия была всегда доступна и убрать это число посредством **DROP** после выхода из цикла.

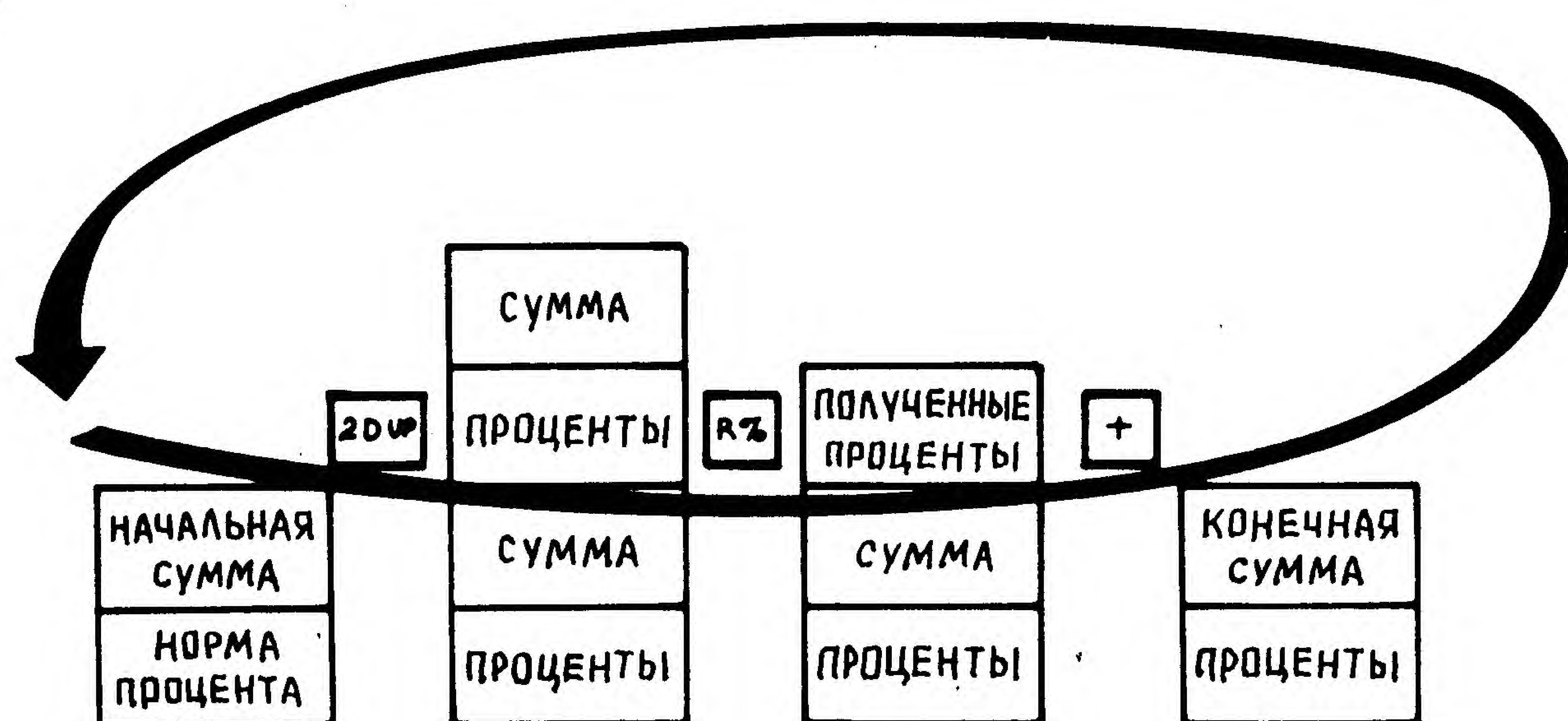
Существует несколько «хитрых» примеров работы со стеком внутри цикла **DO**. Рассмотрим их на примере начисления сложных процентов. При заданном начальном остатке, скажем 1000 дол.

и норме процента, допустим 6%, требуется написать определение для создания и вывода на печать таблицы, приведенной ниже:

```
1000 6 СЛОЖНЫЕ-ПРОЦЕНТЫ
Год 1 Сумма 1060
Год 2 Сумма 1124
Год 3 Сумма 1191
                          и т.д.
```

и т. д. на двадцать лет. Сначала мы загружаем слово R%, специфицированное в гл. 5, а затем создаем определение

```
: СЛОЖНЫЕ-ПРОЦЕНТЫ   ( вклад процент - )
  SWAP  21 1 DO
    CR ." Год "  I .  3 SPACES  2DUP R% + DUP ." Сумма " .
  LOOP  2DROP ;
```



На каждом шаге выполнения цикла мы применяем операцию **2DUP** и таким образом обеспечиваем значение текущего остатка и неизменной нормы процента для следующего шага. После выхода из цикла по окончании вычислений мы убираем эти значения с помощью операции **2DROP**.

Индекс может выступать в качестве некоторого условия для оператора **IF**. С учетом этой возможности вы можете предпринимать конкретные действия только на определенных шагах цикла, например:

```
: ПРЯМОУГОЛЬНИК   256 0 DO   I 16 MOD  0= IF
   CR THEN  ." *"  LOOP ;
```

Слово ПРЯМОУГОЛЬНИК выведет на печать 256 звездочек, причем после каждой 16-й звездочки он будет осуществлять возврат каретки на вашем терминале. В результате вы получите

```
****************
****************
****************
****************
****************
****************
****************
****************
****************
****************
****************
****************
****************
****************
****************
****************
```

ВЛОЖЕННЫЕ ЦИКЛЫ

Ранее мы определили слово с именем ПРОИЗВЕДЕНИЯ, в котором содержался цикл **DO**. При желании можно было бы поместить слово ПРОИЗВЕДЕНИЯ внутрь другого цикла **DO**, например:

```
: ТАБЛИЦА   CR 11 1 DO I ПРОИЗВЕДЕНИЯ  LOOP ;
```

В результате мы получили бы таблицу умножения в таком виде:

```
1  2  3  4  5  6  7  8  9  10
2  4  6  8  10  12  14  16  18  20
3  6  9  12  15  18  21  24  27  30
                                          и т.д.
10  20  30  40  50  60  70  80  90  100
```

так как **I** во внешнем цикле обеспечивает аргумент для слова ПРОИЗВЕДЕНИЯ.

Циклы тоже можно вкладывать один в другой только в пределах одного определения:

```
: ТАБЛИЦА   CR 11 1 DO
     11 1 DO I J *  4 .R  LOOP CR  LOOP ;
```

Обращаем ваше внимание на тот случай, когда во внутреннем цикле находится фрагмент

```
I J *
```

Если **I** копирует индекс того цикла, внутри которого само расположено, то слово **J** копирует индекс *следующего внешнего цикла*. Поэтому выражение I J * приведет к перемножению двух индексов. Таким образом мы создаем значения нашей таблицы.

Что можно сказать теперь о выражении

```
4 .R
```

Это всего лишь видоизмененный оператор, который используется для печати чисел в табличной форме с выравниванием их по вертикали. Цифра 4 определяет заданное число позиций в столбце. Новую таблицу можно теперь распечатать так:

```
1    2    3    4    5    6    7    8    9   10
2    4    6    8   10   12   14   16   18   20
3    6    9   12   15   18   21   24   27   30  и.д.
```

Под каждое число выделено четыре позиции независимо от того, сколько цифр содержится в данном числе. Слово **.R** означает «вывод числа, выравненного вправо».

+ LOOP

Если вы хотите, чтобы значение индекса на каждом шаге выполнения цикла изменялось не на единицу, а на некоторую другую величину, то вместо **LOOP** можно использовать слово **+ LOOP**. Для этого слова в стеке должно находиться число, значение которого является приращением индекса на каждом шаге. Например, в определении

```
: ЧЕРЕЗ-ПЯТЬ   50 0 DO  I .  5 +LOOP ;
```

индекс всякий раз будет увеличиваться на пять, что приведет к следующему результату:

```
ЧЕРЕЗ-ПЯТЬ 0 5 10 15 20 25 30 35 40 45 ок
```

(Представляете, о каких гигантских шагах идет речь.)

Отрицательное приращение заставляет цикл выполняться в обратном направлении, например:

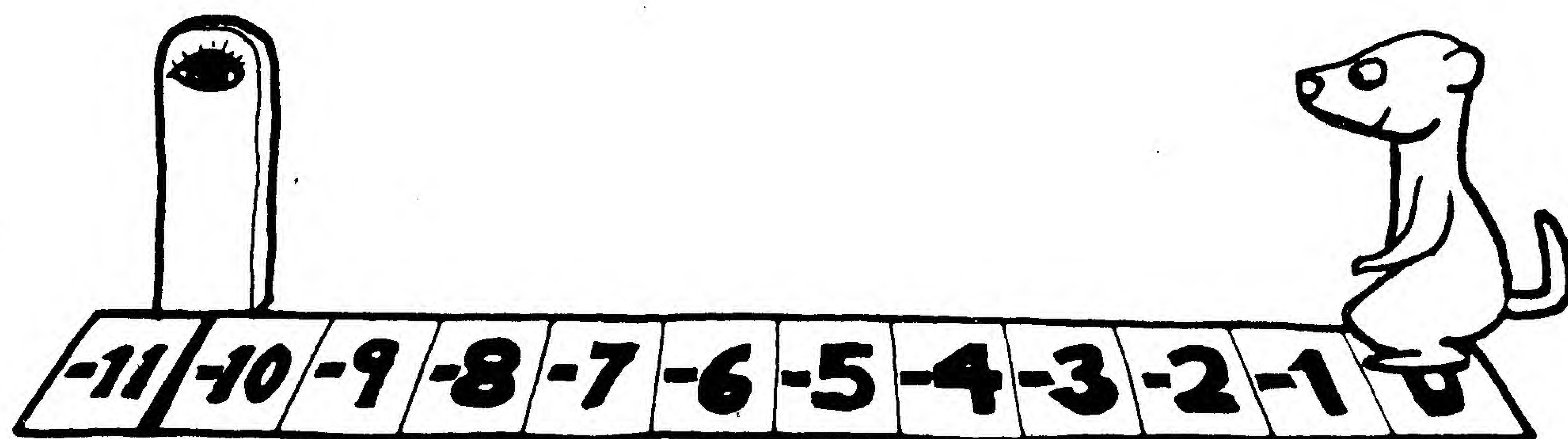
```
: СНИЖЕНИЕ    -10 0 DO I .  -1 +LOOP ;
```

дает такой результат:

```
СНИЖЕНИЕ 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 ок
```

Слово **+ LOOP** начнет с нуля и с шагом, равным —1, «пойдет» к финишной черте, пока не пересечет ее. Заметьте, что в этом направлении мы выполняем цикл фактически 11 раз, поскольку финишная черта всегда лежит между значением границы и ее значе-

нием −1 независимо от того, в какую сторону перемещается **+LOOP**. Завершение же цикла вызывает переход финишной черты.



Приращение может быть получено каким угодно образом, но на каждом шаге выполнения оно должно находиться в стеке. Рассмотрим пример:

```
: ПРИРАЩЕНИЕ  ( приращение граница индекс -- )
   DO I .  DUP +LOOP  DROP ;
```

Внутри этого определения непосредственно нет приращения. Оно будет взято из стека при выполнении слова ПРИРАЩЕНИЕ наряду с границей и индексом. Посмотрите, что происходит в данном случае:

```
Шаг   1 5 0 ПРИРАЩЕНИЕ 0 1 2 3 4 ok

Шаг   2 5 0 ПРИРАЩЕНИЕ 0 2 4 ok

Шаг   -3 -10 10 ПРИРАЩЕНИЕ 10 7 4 1 -2 -5 -8 ok
```

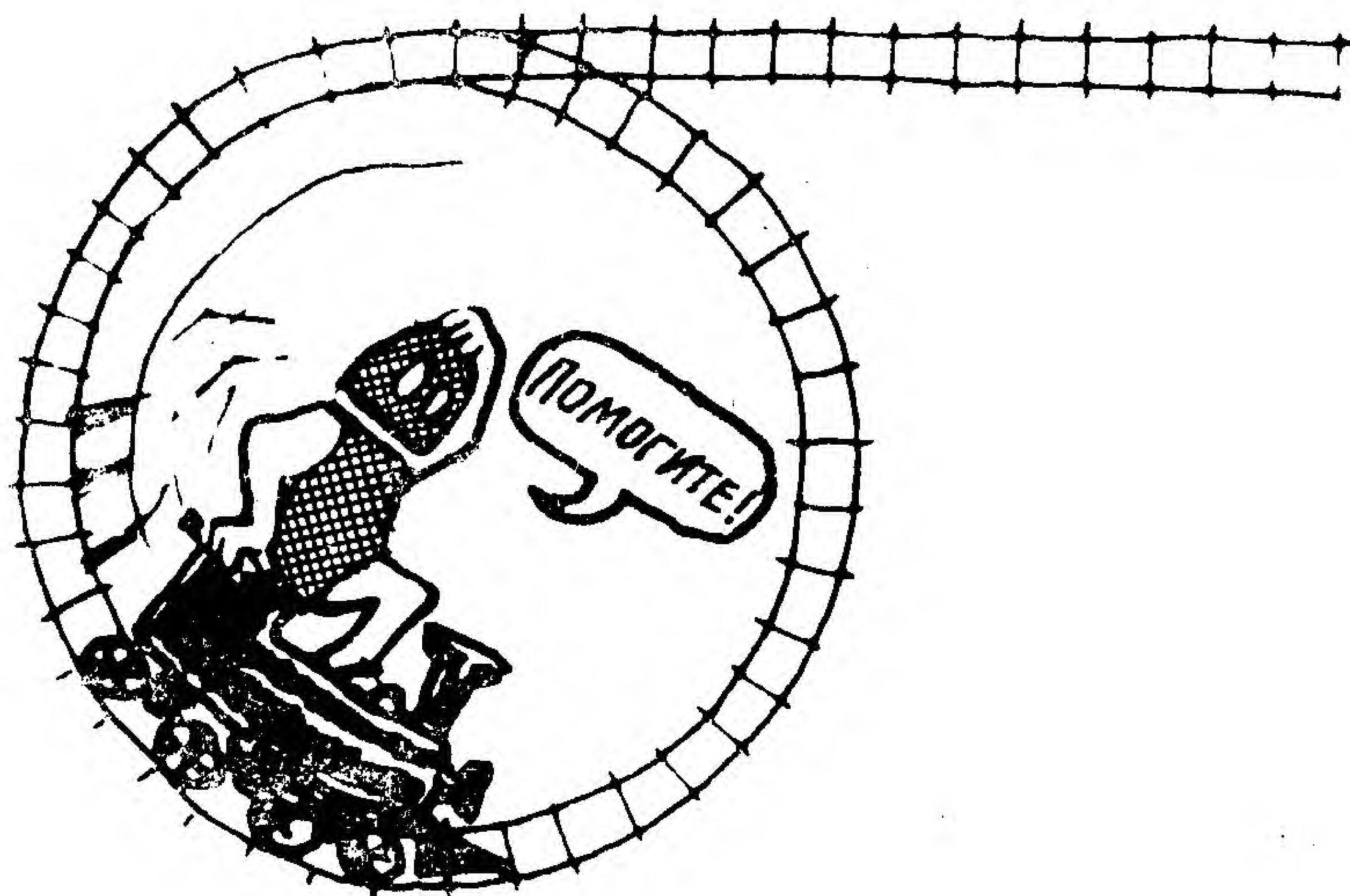
Наш следующий пример демонстрирует изменение приращения на каждом шаге выполнения цикла:

```
: УДВАИВАНИЕ   CR  32767 1 DO  I .  I +LOOP ;
```

Здесь индекс непосредственно используется в качестве приращения. Начиная с единицы он всякий раз удваивается, как это показано ниже:

```
УДВАИВАНИЕ
1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 ok
```

Если бы в этом примере аргумент для + **LOOP** принял хотя бы один раз значение 0, то мы никогда не вышли бы из цикла — получился бы так называемый *бесконечный цикл*[1].



РЕКОМЕНДАЦИИ ПО ПРИМЕНЕНИЮ ОПЕРАТОРА DO В СТИЛЕ ФОРТА

Как уже отмечалось, слова **DO** и **LOOP** являются операторами управления и, следовательно, должны быть выполнены в пределах одного определения. Это означает, что вы не можете выполнить отладку определений, содержащих цикл, в режиме калькулятора, за исключением тех случаев, когда вы моделируете цикл сами.

Рассмотрим, каким образом «оперившийся» программист может заняться отладкой определения СЛОЖНЫЕ-ПРОЦЕНТЫ (из первого раздела настоящей главы). Прежде чем добавить сообщения .” , программист может кратко записать такой вариант на листе бумаги:

```
: СЛОЖНЫЕ-ПРОЦЕНТЫ    ( вклад процент -- )
   SWAP  21 1 DO  CR I .  2DUP  R% +  DUP. LOOP  2DROP ;
```

Он может вести отладку этого варианта за терминалом, используя . или **.S** для проверки результата на каждом шаге цикла. Такой «диалог» выглядит следующим образом:

---

[1] *Для специалистов*. Некоторые Форт-системы, созданные до принятия Стандарта-83, включают /**LOOP**, которое, как и +**LOOP**, выбирает из стека приращение, но оно должно быть положительным. Это может привести к тому, что индекс превысит значение 32767, скажем, при индексации по адресам или номерам блоков, так как указанные числа принимают только положительные значения в отличие от тех ситуаций, когда выполняются операции над числами со знаком. Слово + **LOOP**, удовлетворяющее Стандарту-83, снимает данную проблему.

При моделировании мы не используем фразы «граница счетчик DO», „I.“ и „DUP“.

```
1000 6 SWAP  .S<return>
6 1000 ok
2DUP .S<return>                       Шаг 1
6 1000 6 1000 ok
R% .S<return>
6 1000 60 ok
+ .S<return>
6 1060 ok
2DUP R% +  .S<return>
6 1124 ok
2DROP .S<return>                      Шаг 2
Стек пуст ok
```

Цикл отлажен. Считая, что выполнен последний шаг, проверяем, не пуст ли стек.

Полезный прием. Как очистить стек и при этом эмоционально разрядиться. Иногда начинающий программист может случайно написать цикл, который после себя оставляет в стеке множество значений, например:

```
: ПЯТЕРКИ  ( -- )  100 0 DO  I 5 .  LOOP ;
```

вместо

```
: ПЯТЕРКИ  ( -- )  100 0 DO  I 5 *  .  LOOP ;
```

Если вы увидите, что кто-то попал в такую ситуацию (с вами-то, конечно, подобное никогда не произойдет), и, для того чтобы очистить стек, набирает бесконечную последовательность точек, посоветуйте ему не делать этого, а несколько раз наугад ударить по клавиатуре и нажать клавишу возврата каретки. В результате получится что-нибудь типа

```
ASDFKJ
```

что, естественно, не является словом Форта и приведет к тому, что текстовый интерпретатор вызовет **ABORT**, который, помимо всего прочего, почистит оба стека. (Но объясните новичку, что не нужно бить по клавишам при каждой ошибке.)

ЦИКЛЫ С УСЛОВИЕМ

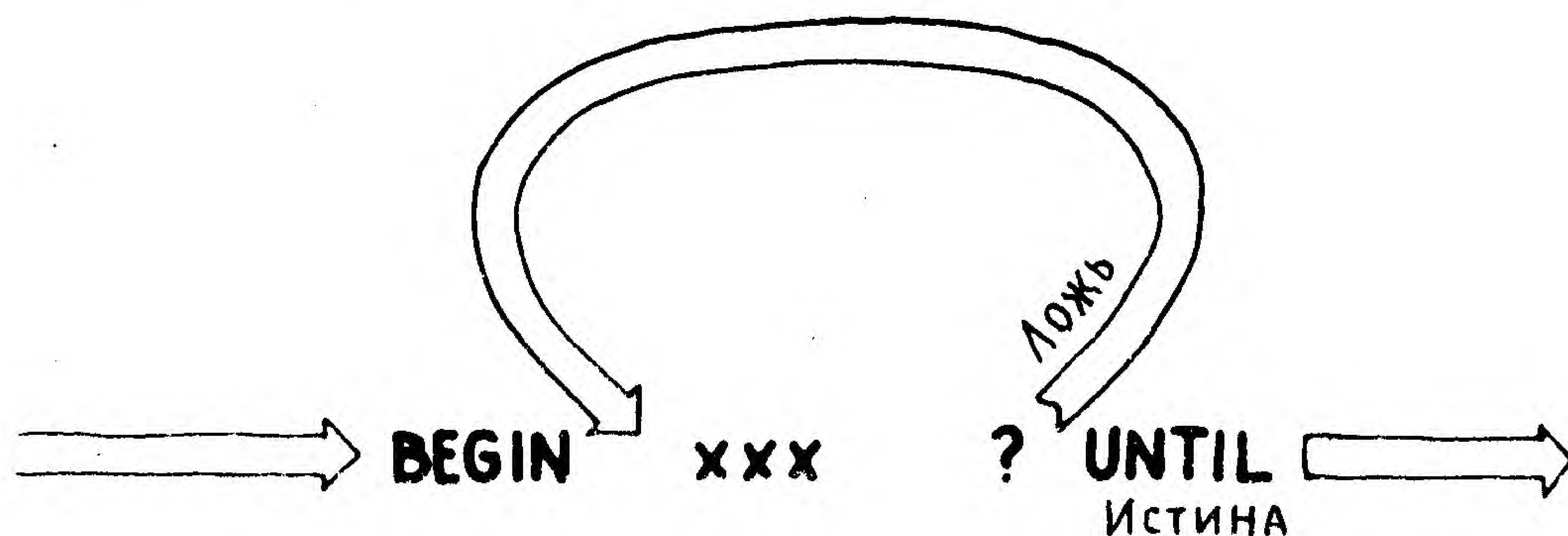
Наряду с циклами **DO**, которые называются циклами со счетчиком, Форт поддерживает *циклы с условием*. Цикл такого рода выполняется либо неопределенно долго, либо до тех пор, пока не произойдет некоторое событие. Один из видов цикла с условием —

```
BEGIN ... UNTIL
```

Этот цикл выполняется до тех пор, пока некоторое условие истинно. Форма его применения:

```
BEGIN  xxx  ? UNTIL
```

Здесь xxx обозначает слова, которые должны повторяться в рамках цикла, а ? — некоторый флаг. Пока флаг принимает значение «ложь», цикл будет повторяться, но как только флаг примет значение «истина», цикл завершится.



Оператор цикла **BEGIN ... UNTIL** используется в одном из определений в примере со стиральной машиной, который рассматривался ранее:

```
: ДО-НАПОЛНЕНИЯ  BEGIN  ?ПОЛОН UNTIL ;
```

где данное определение применяется в другом определении более высокого уровня:

```
: НАЛИТЬ-ВОДУ  КРАНЫ ОТКРЫТЬ  ДО-НАПОЛНЕНИЯ  КРАНЫ ЗАКРЫТЬ ;
```

Слово ?ПОЛОН будет определено таким образом, чтобы осуществлять с помощью электронной схемы проверку индикатора уровня воды в емкости машины, который сигнализирует о заполнении емкости, выдавая значение «истина» в стеке. Слово ДО-НАПОЛНЕНИЯ периодически осуществляет проверку снова и снова (тысячи раз в секунду) до тех пор, пока индикатор, наконец, не подаст свой сигнал, после чего цикл завершится. Затем слово ; в ДО-НАПОЛНЕНИЯ направит на выполнение оставшиеся слова в НАЛИТЬ-ВОДУ, и вода будет перекрыта.

Иногда программист вынужден создавать бесконечный цикл. На Форте это лучше всего сделать следующим образом:

```
BEGIN  xxx  FALSE UNTIL
```

(Слово **FALSE** выполняет те же функции, что и число 0, поэтому, если в вашей системе нет этого слова, используйте вместо него 0) Так как флаг, относящийся к **UNTIL**, всегда имеет значение

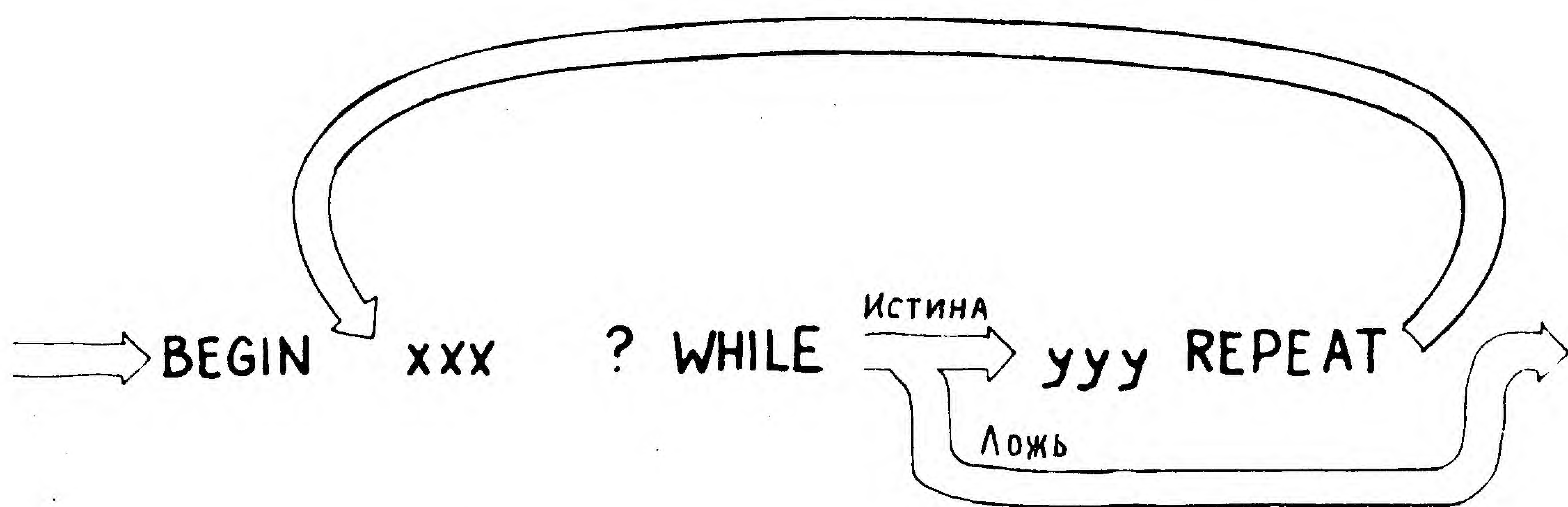
«ложь», цикл будет повторяться бесконечно. В некоторых системах слово **AGAIN** заменяет выражение FALSE UNTIL.

Новички обычно стараются избегать бесконечных циклов, поскольку вход в такой цикл означает, что они потеряли управление над компьютером (в том смысле, что при этом выполняются только слова, заключенные внутри цикла). Но бесконечные циклы имеют право на существование. Например, интерпретатор текста является частью бесконечного цикла под названием **QUIT**. Он ожидает поступления входных данных, интерпретирует и обрабатывает их, выводит "ok", а затем ожидает ввода следующей порции входных данных. В большинстве устройств, управляемых с помощью микропроцессоров, определения самого высокого уровня содержат бесконечный цикл, описывающий работу данного устройства.

Известна еще одна форма организации цикла с условием[1]:

```
BEGIN   xxx   ? WHILE   yyy   REPEAT
```

В таком варианте проверка на выход из цикла осуществляется в теле цикла, а не в его конце. Если условие проверки истинно, то оставшаяся часть цикла выполняется, а затем происходит возврат в начало цикла. Если же результат проверки ложный, цикл заканчивается.



Заметьте, что результат проверки условия в цикле описываемой конструкции противоположен результату проверки в цикле **BEGIN ... UNTIL**. В первом случае цикл повторяется до тех пор, *пока некоторое условие истинно* (а не до тех пор, *пока оно не станет истинным*).

Предположим, у нас имеется некоторая база данных, заполненная именами. Мы уже определили слово НОМЕР-ИМЕНИ, помещающее в стек либо номер следующего имени из списка, либо

[1] REPEAT — ПОВТОРЯТЬ, WHILE — ПОКА. — *Примеч. пер.*

значение «ложь», если список исчерпан. Кроме того, у нас есть слово .ИМЯ, которое выводит само имя по его номеру в стеке.

```
: НОМЕР-ИМЕНИ  ( -- номер-имени-или-ложь )  ... ;
: .ИМЯ  ( номер-имени -- )  ... ;
```

Допустим, нам нужно вывести список всех имен. До тех пор, пока не распечатан список, мы не знаем, сколько там имен, и поэтому не можем воспользоваться конструкцией **DO LOOP**, но можем написать определение:

```
: .ВСЕ-ИМЕНА
   BEGIN  НОМЕР-ИМЕНИ DUP WHILE  CR .ИМЯ  REPEAT  DROP ;
```

А что произойдет, если список окажется пустым? На первом шаге выполнения цикла НОМЕР-ИМЕНИ поместит в стек нуль и вторая часть цикла никогда не будет исполнена. Слова **WHILE** исключает необходимость в специальной проверке.

ВЫХОД ИЗ ЦИКЛА (LEAVE) И ВЕТВЛЕНИЕ (BRANCH)

Существует способ написания цикла со счетчиком, при котором выполнение цикла может закончиться раньше, чем будет достигнуто заданное значение границы. Для этого нужно с помощью слова **LEAVE** запрограммировать внутри цикла **DO** изменение условия с «истины» на «ложь». **LEAVE** заставляет цикл немедленно завершиться[1].

Перепишем наше прежнее определение слова СЛОЖНЫЕ-ПРОЦЕНТЫ. Теперь мы не будем запускать цикл ровно 20 раз, а организуем завершение этого цикла либо после его 20-го выполнения, либо после удвоения денежной суммы — в зависимости от того, какое из событий произойдет раньше.

Добавим следующую фразу:

```
2000 > IF LEAVE THEN
```

как в следующем тексте:

```
: УДВОЕНО  ( вклад процент -- )
   SWAP  21 1 DO
      CR ." Год "  I  2 .R  3 SPACES 2DUP  R% + DUP
      ." Сумма "  .  DUP 2000 > IF
         CR ." Более чем удвоено через " I .  ." лет " LEAVE
      THEN
   LOOP 2DROP ;
```

---

[1] *Для пользователей систем, разработанных до принятия Стандарта-83.* Ранее слово **LEAVE** вызывало завершение цикла при выполнении очередного слова **LOOP** или **+LOOP**.

В результате получим:

```
1000 6 УДВОЕНО

Год  1   Сумма 1060
Год  2   Сумма 1124
Год  3   Сумма 1191
Год  4   Сумма 1262
Год  5   Сумма 1338
Год  6   Сумма 1418
Год  7   Сумма 1503
Год  8   Сумма 1593
Год  9   Сумма 1689
Год 10   Сумма 1790
Год 11   Сумма 1897
Год 12   Сумма 2011

Более чем удвоено через 12 лет
```

В одном из упражнений в конце главы вам предлагается переработать слово УДВОЕНО таким образом, чтобы оно выбирало из стека в виде аргументов норму процента и начальную сумму и выполняло бы вычисления до получения удвоенной начальной суммы, после чего слово **LEAVE** завершало бы эти вычисления.

Слово **LEAVE** ведет себя, как и положено при использовании его во вложенных операторах цикла **DO**: оно просто «покидает» цикл, в котором находится. В одном цикле может употребляться несколько слов **LEAVE**. Цикл завершается при встрече первого из них.

Согласно Стандарту-83, любой код, размещенный после сочетания IF LEAVE THEN, *не будет* исполнен на последнем шаге. Как правило, лучше применять это сочетание в качестве последнего выражения перед **LOOP**.

Еще одно предупреждение: как и все условные операторы, слово **LEAVE** должно находиться в том же определении, что и условия, по которым оно покидает цикл (**DO** и **LOOP**). Запомните, что нельзя помещать **LEAVE** в определении слова, вызываемого из цикла. Приведем пример неправильного использования слова **LEAVE**:

```
: ВЫБОР   ВАРИАНТ1 IF ВАРИАНТ2  LEAVE  THEN ;
: НЕПРАВИЛЬНЫЙ-ЦИКЛ   1000 0 DO  ВЫЧИСЛЕНИЯ  ВЫБОР  LOOP ;
```

Здесь **LEAVE** используется вне цикла. Эти определения будут скомпилированы, но при выполнении приведут к непредсказуемому результату (возможно, даже к разрушению системы).

Два полезных приема: PAGE и QUIT. Для того чтобы придать более аккуратный вид данным, выводимым циклически (таким как таблицы и геометрические фигуры), перед выводом информации вам, возможно, придется очистить экран

с помощью слова **PAGE** (СТРАНИЦА). Вы можете использовать слово **PAGE** непосредственно всякий раз, когда нужно очистить экран.

```
PAGE ПРЯМОУГОЛЬНИК
```

При этом экран будет очищаться перед выводом прямоугольника, который вы определили ранее. А можно поместить слово **PAGE** один раз в начало определения:

```
: ПРЯМОУГОЛЬНИК   PAGE  256 0 DO
   I 16 MOD 0=  IF CR THEN  ." *" LOOP ;
```

Если вы не хотите, чтобы по завершении вычисления на экране появилось приглашение ok, примените слово **QUIT** (ВЫЙТИ). Вы можете использовать **QUIT** непосредственно:

```
ПРЯМОУГОЛЬНИК QUIT
```

а можете сделать его последним словом определения (перед точкой с запятой).

Ниже дается перечень слов Форта, приводимых в настоящей главе.

DO ... LOOP	DO: (граница индекс --) LOOP: (--)	Организация цикла со счетчиком по заданному диапазону индексов.
DO ... +LOOP	DO: (граница индекс --) +LOOP: (n --)	Аналогично DO ... LOOP . Только к индексу на каждом шаге добавляется значение n (а не как всегда единица).
LEAVE	(--)	Немедленное завершение выполнения цикла LOOP или + LOOP. (Используется только внутри цикла.)
BEGIN ... UNTIL	UNTIL: (? --)	Организация цикла с условием, который завершается, когда ? принимает значение "истина".
BEGIN xxx WHILE yyy REPEAT	WHILE: (? --)	Организация цикла с условием, причем xxx выполняется всегда, а yyy–только если ? ложно.
.R	(u ширина-поля --)	Вывод числа одинарной точности без знака. Число выровнено справа по границе поля.
PAGE	(--)	Чистка экрана дисплея и установка курсора в верхний левый угол.
QUIT	(--)	Завершение выполнения текущей задачи и возврат управления на терминал.

ОСНОВНЫЕ ТЕРМИНЫ

Бесконечный цикл. Циклическая структура, позволяющая повторять слова, содержащиеся внутри цикла, без возможности внешнего прерывания. Остановить такие вычисления можно только отключением или перезагрузкой компьютера.

Цикл со счетчиком. Циклическая структура, позволяющая повторять слова, содержащиеся внутри цикла, определенное число раз. В Форте это число зависит от начального и конечного счетчиков (индекса и границы), которые помещаются в стек до выполнения слова **DO**.

Цикл с условием. Циклическая структура, при которой слова, содержащиеся внутри цикла, повторяются до тех пор, пока не изменится некоторое условие истинности (с истины на ложь или со лжи на истину). В Форте бесконечный цикл начинается со слова **BEGIN**.

УПРАЖНЕНИЯ

В задачах с 6.1 и по 6.6 вы должны создать определения нескольких слов, которые будут выводить шаблоны из звездочек. Это потребует применения циклов **DO** и **BEGIN ... UNTIL**.

6.1. Создайте слово с именем STARS, которое выводит n звездочек в одной строке. n задается в стеке.

6.2. Определите слово БРУСОК, которое выводит прямоугольник из звездочек. Ширина и высота (число строк) задаются в стеке в следующем порядке: (ширина высота --).

```
10 3 БРУСОК
**********
**********
********** ok
```

6.3. Создайте слово \STARS, которое выводит наклонные (ромбовидные) массивы из звездочек. Высота задается в стеке. Используйте цикл **DO** и для простоты примите ширину постоянной и равной 10 звездочкам.

```
3 \STARS
 **********
  **********
   ********** ok
```

6.4. Определите слово, которое выводит массивы из звездочек с наклоном в другую сторону, и назовите его /STARS. Высота должна задаваться в стеке, а ширину возьмите постоянной и равной 10 звездочкам. Используйте оператор **DO**.

6.5. Переопределите последнее слово с использованием оператора цикла **BEGIN ... UNTIL**.

6.6. Напишите определение с именем РОМБЫ, которое выводит заданное число ромбов, как показано в следующем примере:

6.7 В гл. 3 было введено слово **THRU**. Сможете ли вы написать определение этого слова с помощью вызова **LOAD**? Если хотите, расширьте функции слова так, чтобы при использовании его во время загрузки блока выводился бы номер последнего.

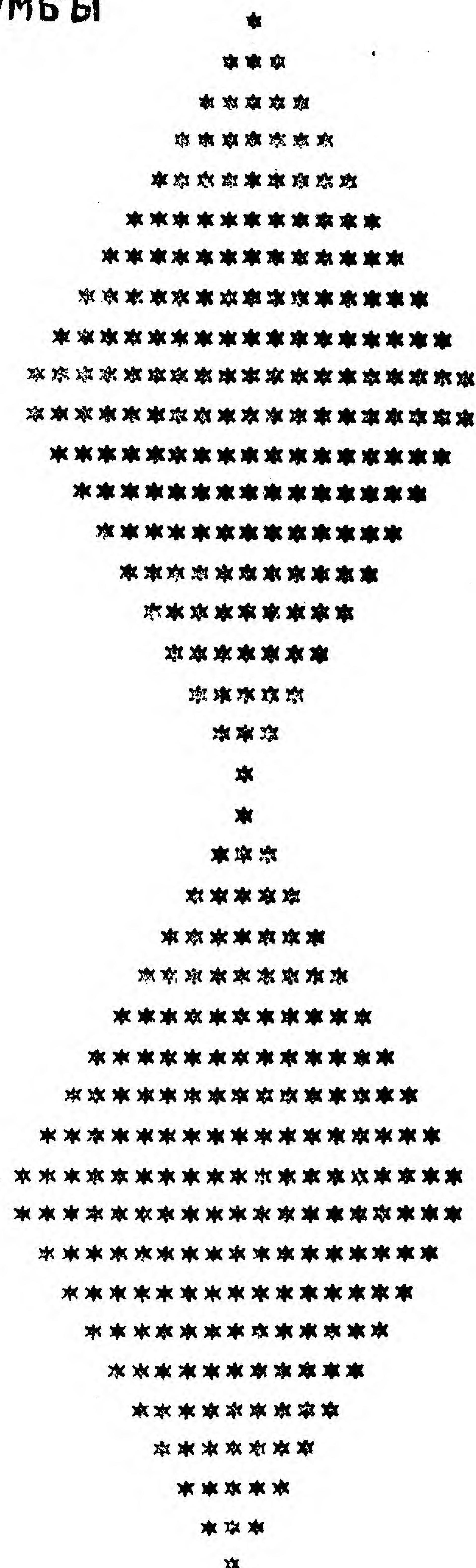
6.8. При рассмотрении слова **LEAVE** мы приводили пример, в котором вычислялись сложные проценты при норме процента 6% и начальном остатке 1000 дол. за 20 лет или же до тех пор, пока начальный остаток не удвоится, в зависимости от того, какое из событий наступит раньше. Перепишите это определение таким образом, чтобы значения начальной суммы и нормы процента находились в стеке и процесс вычисления завершился с помощью **LEAVE** как только сумма начального остатка удвоится.

6.9. Определите слово с именем **, которое вычисляло бы степень, например:

```
7 2 ** . 49 ok
( семь в квадрате )

2 4 ** . 16 ok
( два в четвертой степени )
```

Для простоты показатель степени примите положительным, но убедитесь в том, что ** выполняется правильно, если показатель равен нулю (в результате должна получиться единица) или единице (в результате должно получиться само число).



Глава 7
ЧИСЛО ТИПОВ ЧИСЕЛ

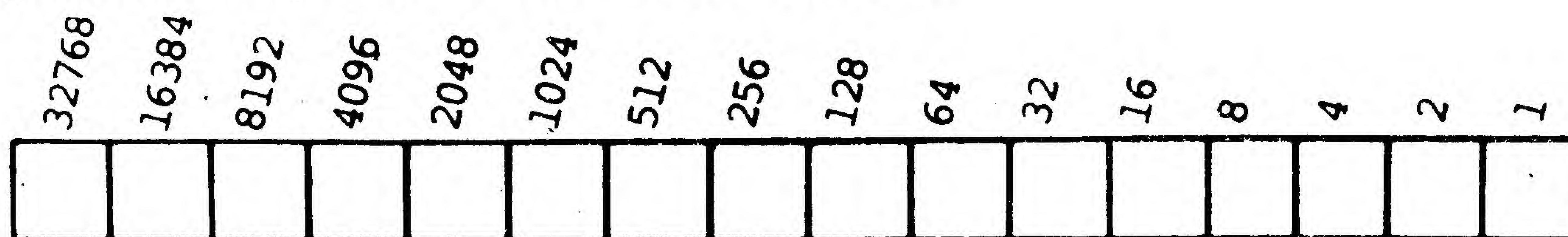
До сих пор речь шла только о числах одинарной длины со знаком. В настоящей главе мы введем числа без знака и числа двойной длины и познакомим вас с новыми операциями над этими числами.

Излагаемый здесь материал разделен на две части. В первой части (для начинающих) рассматриваются числа применительно к компьютеру, а также дается точное определение терминов *со знаком, без знака, одинарная длина, двойная длина,* во второй (для всех) продолжается обсуждение Форта и объясняется, как трактуются числа со знаком и без знака, одинарной и двойной длины (точности).

Часть 1
ДЛЯ НАЧИНАЮЩИХ

ЧЕМ ОТЛИЧАЮТСЯ ЧИСЛА СО ЗНАКОМ И БЕЗ ЗНАКА

Все цифровые компьютеры хранят числа в двоичной форме[1]. Элемент стека Форт-системы состоит из 16 разрядов[2], или битов (бит — двоичная цифра). Ниже показана структура из 16 битов, где приводятся значения всех битов:



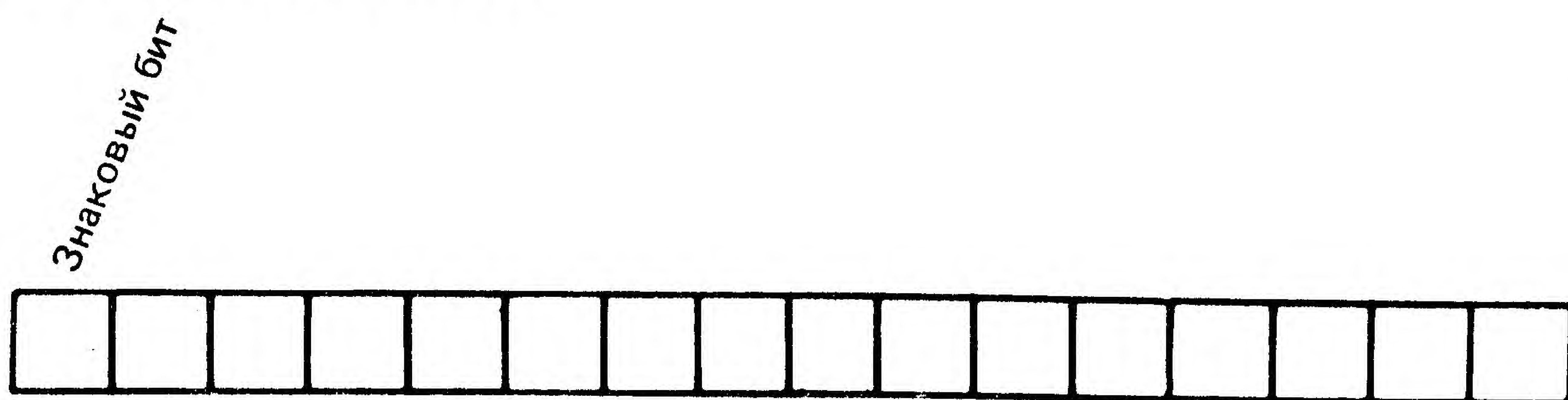
Если в каждом бите хранится единица, то это число составит 65535. Таким образом, 16 битами можно представить любое зна-

[1] *Для тех, кто не знаком с двоичной системой счисления.* Попросите кого-нибудь из ваших друзей (увлекающегося математикой) рассказать вам об этой системе или поищите учебное пособие по ЭВМ для начинающих.

[2] *Для пользователей 32-разрядных компьютеров.* В таких процессорах, как 68000, стек Форта обычно 32-разрядный. Поэтому термин, *число одинарной длины* в данном случае означает 32-разрядное число.

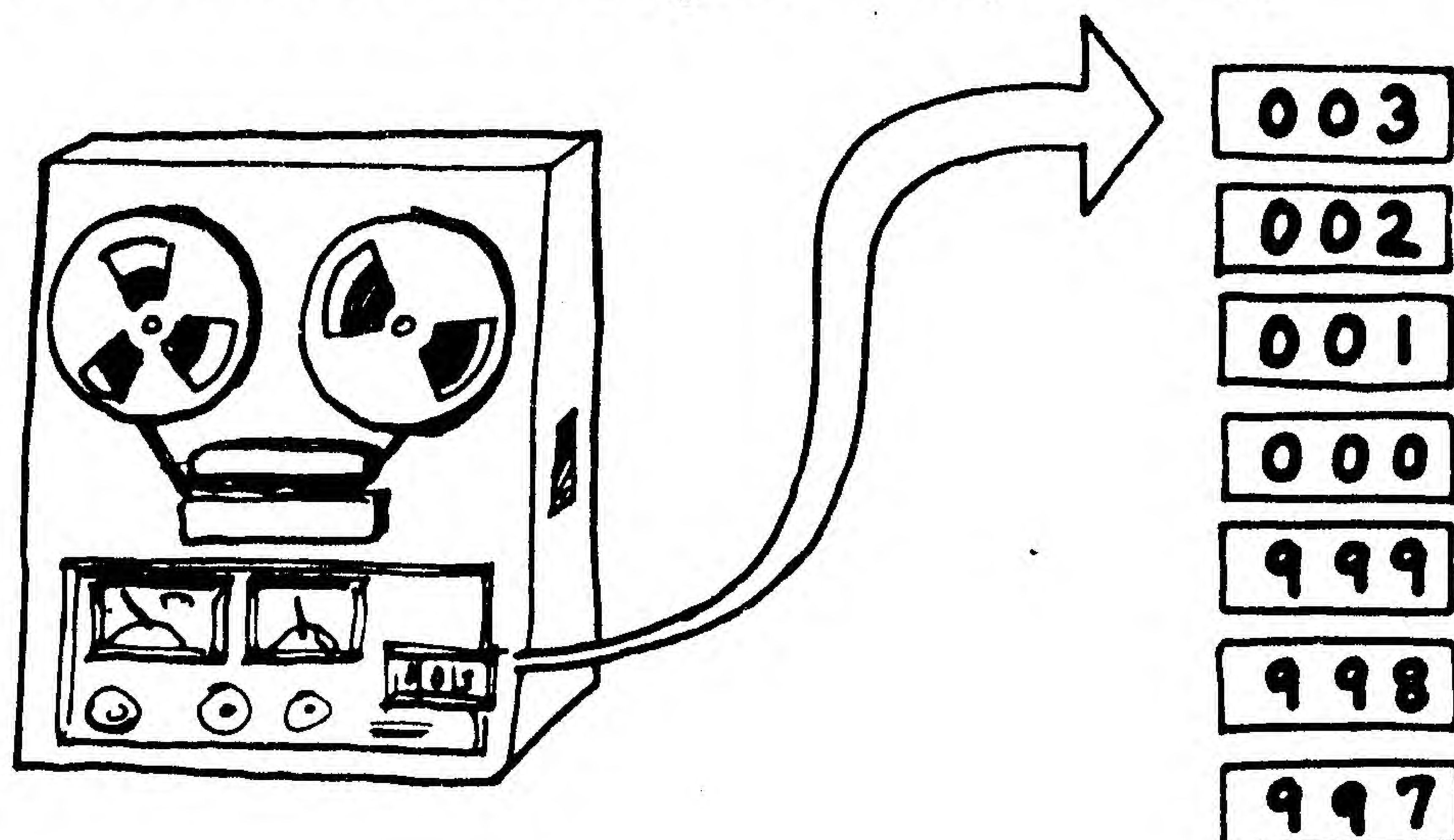
чение от 0 до 65535. Поскольку такой способ представления не позволяет нам изображать отрицательные числа, мы называем их *числами без знака*. В наших таблицах и стековой нотации числа без знака обозначаются буквой *u*.

Но как же быть с отрицательными числами? Чтобы иметь возможность изображать как положительные, так и отрицательные числа, необходимо пожертвовать одним битом, задействовав его под знак. Для этой цели мы отведем самый левый «старший по порядку» бит. 15 оставшихся бит позволят нам представить любое число вплоть до 32767. Если знаковый бит содержит единицу, то нужно отложить изображаемое значение по оси влево от нуля, т. е. в области отрицательных чисел. Таким образом, с помощью 16 бит можно представить число в диапазоне от −32768 до +32767. Вы уже знаете, что этот диапазон составляют числа одинарной длины, обозначаемые буквой *n*.



Мы столь подробно останавливаемся на представлении отрицательных чисел, чтобы у вас была полная ясность по всем излагаемым здесь вопросам. У вас может сложиться впечатление, что достаточно просто ввести знаковый разряд, и вы сможете различить отрицательное и положительное числа, однако на самом деле это далеко не так.

Чтобы понять, как представляются отрицательные числа, вернемся снова к десятичной системе счисления и понаблюдаем за показаниями счетчика, устанавливаемого на магнитофонах многих типов. Допустим, что счетчик высвечивает три цифры (трехзначное число). По мере перемотки ленты это число увеличивается. Установим счетчик на нуль и начнем перематывать ленту в обратном направлении. Первое число, которое вы увидите на индикаторе, будет 999. Его мы воспринимаем как −1. Следующим числом окажется 998, что соответствует −2, и т. д.



Аналогично представляются числа со знаком и в компьютере. Если мы начнем с нуля:
0000000000000000
и вернемся назад на одно число, то будем иметь
1111111111111111 (16 единиц)
что означает 65535 при изображении чисел без знака и —1 при изображении чисел со знаком. Число
1111111111111110
соответствует 65534 при изображении чисел без знака и —2 при изображении чисел со знаком.

(Традиционно числа —1 и 0 применяются в качестве значений «истина» и «ложь» потому, что в представлении —1 все биты установлены, а в представлении 0 — сброшены.)

Ниже приводится схема, в которой показано, каким образом двоичное число в стеке может использоваться как число без знака или как число со знаком.

Как число без знака		Как число со знаком
65535	1111111111111111	
...	...	
32768	1000000000000000	
32767	0111111111111111	32767
...	...	...
0	0000000000000000	0
	1111111111111111	-1
	...	...
	1000000000000000	-32768

Такой странный на первый взгляд способ представления отрицательных чисел дает возможность компьютеру использовать одни и те же процедуры для выполнения как вычитания, так и сложения.

Продемонстрируем изложенное на примере простой задачи:

  2
—1
—

Эта задача эквивалентна задаче сложения 2+(—1). При двоичном представлении чисел одинарной длины двойка выглядит следующим образом:
0000000000000010
а отрицательная единица — так:
1111111111111111

Компьютер складывает их таким же образом, как мы это делаем на бумаге. Если сумма в какой-то колонке превышает единицу, то в следующую колонку (старший двоичный разряд) переносится 1. В результате получается:

```
+ 0000000000000010
  1111111111111111
  ----------------
  1000000000000001
```

Как видите, компьютеру потребовалось выполнить перенос единицы в старший разряд во всех колонках, а последняя единица была вынесена в 17-й разряд. Но так как элемент стека состоит только из 16 битов, будет выведено число
0000000000000001
т. е. единица, что является правильным ответом.

Мы не объясняем здесь, каким образом компьютер переводит положительное число в отрицательное. При желании вы можете найти описание этого процесса, который называется «дополнением до двух», в литературе.

АРИФМЕТИЧЕСКИЙ СДВИГ

Когда мы рассматривали в гл. 5 выполнение компьютером некоторых арифметических операций, нам встретились две «таинственные» фразы: «арифметический сдвиг влево» и «арифметический сдвиг вправо». Теперь настала пора объяснить их.

От чего зависит быстрота ответа Форт-системы.

```
2*          ( n -- n*2)       Умножение на два ( арифметический
                              сдвиг влево ).

2/          ( n -- n/2)       Деление на два ( арифм_етический
                              сдвиг вправо ).
```

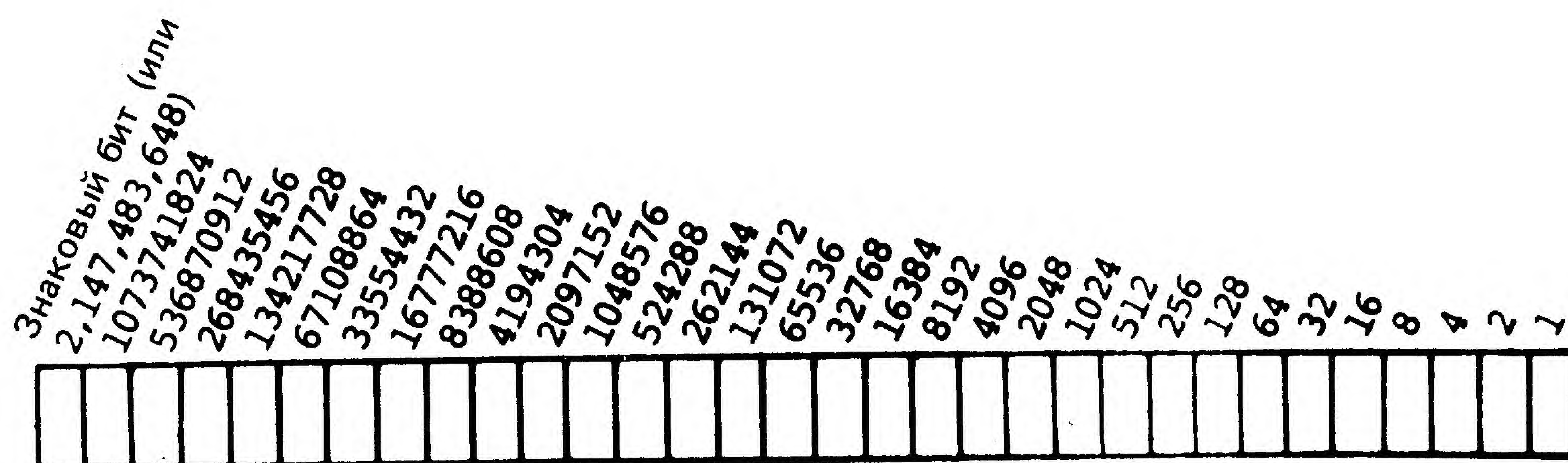
Для иллюстрации представим какое-нибудь число, скажем шесть, в двоичном виде:
0000000000000110
(4 + 2). Сдвинем каждую цифру этого числа на один разряд влево, а освободившийся бит заполним нулем:
0000000000001100

Мы получили двоичное представление числа 12 (8 + 4), что ровно в два раза больше первоначального числа. Такой способ применим во всех случаях. Если вы переместите каждую цифру числа на одну позицию *вправо* и заполните освободившийся бит нулем, то в результате всегда получите *половину* первоначального числа.

При арифметическом сдвиге знаковый разряд не смещается. Иными словами, положительное число остается положительным, а отрицательное — отрицательным при выполнении операций деления и умножения на два. (Сдвиг, при котором вместе с остальными битами сдвигается и старший по порядку бит, называется *логическим сдвигом.*) Вы должны знать, что компьютер выполняет сдвиг цифр намного быстрее, чем операции обычного деления или умножения. Когда скорость критична, то лучше написать 2*, чем 2 *, и даже может быть лучше написать 2* 2* 2*, чем 8 * — все зависит от конкретной модели вашего компьютера.

ЧИСЛА ДВОЙНОЙ ДЛИНЫ

Вы, вероятно, уже знаете, что такое число двойной длины. Это число, представление которого занимает 32 бита вместо 16. Двойная длина допускает представление чисел в диапазоне ±2 147 483 647 (т. е. свыше четырех биллионов).



В Форте число двойной длины занимает в стеке место двух чисел одинарной длины. Такие операции, как **2SWAP** и **2DUP**, применимы и для чисел двойной длины, и для пары чисел одинарной длины.

Обращаем ваше внимание на то, что в программировании термином «слово» принято обозначать 16-разрядное число, или два байта. В Форте же под словом понимают некоторым образом определенную команду. Поэтому, чтобы избежать коллизий, программирующие на Форте 16-разрядное значение называют *ячейкой*. Число двойной длины требует для своего представления двух ячеек.

ПРЕИМУЩЕСТВА ШЕСТНАДЦАТИРИЧНОЙ СИСТЕМЫ СЧИСЛЕНИЯ (И ДРУГИЕ СИСТЕМЫ)

Как только вы начнете серьезно программировать, вам, кроме двоичной и десятичной систем счисления, понадобятся другие системы, особенно шестнадцатиричная (основание 16) и восьмиричная (основание 8). Позднее мы их подробно рассмотрим, а пока вам не помешает небольшое введение.

Программисты начали использовать шестнадцатиричные и восьмиричные числа потому, что компьютеры «думают» в двоичной системе, а человеку очень трудно воспринимать числа, состоящие из длинного ряда двоичных цифр. Намного проще перевести двоичное число в шестнадцатиричное, чем двоичное в десятичное, поскольку 16 является степенью числа 2, в то время как 10 — нет. То же самое справедливо и для восьмиричной системы. Поэтому программисты обычно применяют для записи двоичных чисел, которыми компьютер обозначает адреса и машинные коды, шестнадцатиричную или восьмиричную системы. Шестнадцатиричная система выглядит на первый взгляд необычно, так как в ней задействованы буквы от A до F.

ДЕСЯТИЧНАЯ	ДВОИЧНАЯ	ШЕСТНАДЦАТИРИЧНАЯ
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Возьмем двоичное число одинарной длины: 0111101110100001. Для того чтобы перевести это число в шестнадцатиричную систему, мы сначала должны разбить его на тетрады:

0111 1011 1010 0001

затем заменить каждую тетраду на ее шестнадцатиричный эквивалент:

7 B A 1 или просто 7BA1.

В восьмиричной системе используются только цифры от 0 до 7. Большинство современных компьютеров применяет шестнадцатиричную систему, поэтому мы не станем приводить пример на перевод в эту систему. Перевод чисел из одной системы в другую рассматривается более детально в настоящей главе позднее.

КОД ДЛЯ ПРЕДСТАВЛЕНИЯ СИМВОЛЬНОЙ ИНФОРМАЦИИ (ASCII)

Если числа в компьютере хранятся в двоичной форме, то как в нем хранятся буквы или другие символы? Тоже в двоичной форме, только в специальном коде, который был принят в качестве промышленного стандарта много лет назад, — ASCII[1] (стандартный американский код для обмена информацией). В табл. 7.1 приводятся все символы системы и их цифровые представления как в шестнадцатиричной, так и в десятичной форме.

Символы, помещенные в первом столбце (имеющие шестнадцатиричные коды 0 — 1F), называются *управляющими*, так как они указывают, что́ терминал или компьютер должен выполнить, например подать звуковой сигнал, вернуть каретку на одну позицию, начать новую строку и т. д. Остальные символы называются *печатными*, поскольку они выводят на печать видимые символы, включая буквы, цифры от 0 до 9, все доступные символы и даже пробел (шестнадцатиричный код 20). Единственным исключением является символ DEL (шестнадцатиричный код 7F), который предписывает компьютеру игнорировать последний переданный символ.

В первой главе мы ввели слово **EMIT**, которое берет из стека значение в коде ASCII и посылает его на терминал для распечатки этого значения в виде символа, например:

```
65 EMIT A ok
66 EMIT B ok
```

(Мы используем десятичное, а не шестнадцатиричное представление, потому что ваш современный компьютер больше всего приспособлен ко вводу именно в таком представлении[2].)

Почему бы нам не проверить действие **EMIT** на каждом печатном символе «автоматически»?

```
: ОТОБРАЖАЕМЫЕ   127 32 DO I EMIT SPACE LOOP ;
```

---

[1] Код ASCII совпадает с применяемым в СССР семиразрядным кодом КОИ-7. В реализациях Форта на отечественных ЭВМ используются восьмиразрядные коды (например, КОИ-8) с добавлением кодов русских букв, что и позволяет использовать русские слова. — *Примеч. ред.*

[2] *Для специалистов.* Не тратьте время на раздел для начинающих.

Таблица 7.1.

Символы в коде ASCII и их числовые эквиваленты

Символ	Дес.	Шестн.	Символ	Дес.	Шестн.	Символ	Дес.	Шестн.	Символ	Дес.	Шестн.
NUL	00	0	SP	20	32	@	40	64		60	96
SOH	01	1	!	21	33	A	41	65	a	61	97
STX	02	2	"	22	34	B	42	66	b	62	98
ETX	03	3	#	23	35	C	43	67	c	63	99
EOT	04	4	$	24	36	D	44	68	d	64	100
ENQ	05	5	%	25	37	E	45	69	e	65	101
ACK	06	6	&	26	38	F	46	70	f	66	102
BEL	07	7	'	27	39	G	47	71	g	67	103
BS	08	8	(	28	40	H	48	72	h	68	104
HT	09	9	)	29	41	I	49	73	i	69	105
LF	0A	10	*	2A	42	J	4A	74	j	6A	106
VT	0B	11	+	2B	43	K	4B	75	k	6B	107
FF	0C	12	,	2C	44	L	4C	76	l	6C	108
CR	0D	13	-	2D	45	M	4D	77	m	6D	109
SM	0E	14	.	2E	46	N	4E	78	n	6E	110
SI	0Г	15	/	2F	47	O	4F	79	o	6F	111
DLE	10	16	0	30	48	P	50	80	p	70	112
DC1	11	17	1	31	49	Q	51	81	q	71	113
DC2	12	18	2	32	50	R	52	82	r	72	114
DC3	13	19	3	33	51	S	53	83	s	73	115
DC4	14	20	4	34	52	T	54	84	t	74	116
NAK	15	21	5	35	53	U	55	85	u	75	117
SYN	16	22	6	36	54	V	56	86	v	76	118
ETB	17	23	7	37	55	W	57	87	w	77	119
CAN	18	24	8	38	56	X	58	88	x	78	120
EM	19	25	9	39	57	Y	59	89	y	79	121
SUB	1A	26	:	3A	58	Z	5A	90	z	7A	122
ESC	1B	27	;	3B	59	[	5B	91	{	7B	123
FS	1C	28	<	3C	60	\	5C	92	|	7C	124
GS	1D	29	=	3D	61	]	5D	93	}	7D	125
RS	1E	30	>	3E	62	^	5E	94	~	7E	126
US	1F	31	?	3F	63	_	5F	95	DEL (RB)	7F	127

В первом столбце перечислены символы в коде ASCII или, если это управляющие символы, в общепринятых обозначениях; в двух последующих столбцах даются их шестнадцатиричные и десятичные эквиваленты

Слово ОТОБРАЖАЕМЫЕ выведет на печать каждый требуемый символ из кода ASCII, т. е. символы с кодами от десятичного 32 до десятичного 126. (Мы используем коды ASCII в качестве индекса цикла **DO**.)

```
ОТОБРАЖАЕМЫЕ ! " # ¤ & ' ( ) * + ... ok
```

Начинающие могут поинтересоваться, как поведет себя **EMIT** с управляющими символами; наберите на клавиатуре такой текст:




```
7 EMIT ok
```

Вы услышите какой-то сигнал, который является вариантом звонка печатающей машинки для дисплея. (В некоторых системах слово **EMIT** вместо того чтобы выполнить команду, выводит специфические символы.)

Неплохо знать следующие управляющие символы:

НАЗВАНИЕ	ОПЕРАЦИЯ	ДЕСЯТИЧНЫЙ ЭКВИВАЛЕНТ
BS	Возврат назад на одну позицию ("забой")	8
LF	Перевод строки	10
CR	Возврат каретки	13

Поэкспериментируйте с этими управляющими символами и посмотрите, как они выполняются.

Код ASCII разработан таким образом, что каждый символ в нем может быть представлен одним байтом. В приводимых здесь таблицах буква «с» означает, что содержимое некоторого байта соответствует символу в коде ASCII.

В некоторых Форт-системах имеется слово **ASCII**, которое используется для улучшения читабельности определений, поскольку переводит отдельные символы в шестнадцатиричные значения. Вспомним, к примеру, следующее определение:

```
: STAR   42 EMIT ;
```

Если в вашей системе есть это слово, то вы можете определить его так:

```
: STAR   ASCII * EMIT ;
```

В обоих случаях элементы словаря после трансляции определений будут абсолютно одинаковы, однако последнее определение легче воспринимается.

ASCII	(— — c)	Перевод следующего символа из входного потока в его ASCII-эквивалент

Часть 2

ДЛЯ ВСЕХ

ДВОИЧНАЯ ЛОГИКА

Слова **AND** и **OR** (введенные в гл. 4) используют «двоичную логику», т. е. каждый бит проверяется независимо, и перенос единицы в старший разряд не производится. Выполним, например, операцию **AND** над следующими двумя двоичными числами:

```
 0000000011111111
 0110010110100010   AND
 ----------------
 0000000010100010
```

Для того чтобы результирующий бит был равен единице, соответствующие биты-аргументы должны быть оба равными единице. Заметьте, что в этом примере первый аргумент содержит все нули в старшем байте и все единицы в младшем. Действие второго операнда здесь заключается в том, что младшие восемь битов сохраняются неизменными, а старшие восемь битов сбрасываются в нуль. Первый операнд служит «маской» для маскирования старшего байта второго операнда.

Слово **OR** также применяет двоичную логику. В примере

```
 1000100100001001
 0000001111001000   OR
 ----------------
 1000101111001001
```

единица получается в тех битах, где хотя бы один операнд был равен единице. И снова каждый столбец проверяется независимо, без переноса единицы в старший разряд. При умелом использовании масок в одном 16-разрядном значении можно хранить 16 отдельных флагов. Так, мы можем узнать, чему равен соответствующий (скажем, пятый) бит:
1011101010011100
нулю или единице, путем маскирования остальных флагов:

```
 1011101010011100
 0000000000010000   AND
 ----------------
 0000000000010000
```

Так как значение нашего бита равно единице, результат будет истинным. Если бы значение бита составляло нуль, то результат

оказался бы ложным. Мы можем сбросить определенный флаг в нуль, не трогая остальные, следующим приемом:

```
 1011101010011100
 1111111111101111   AND
─────────────────
 1011101010001100
             ↑
```

Мы работали с маской, во всех битах которой, за исключением сбрасываемого в нуль, содержатся единицы. Можно установить тот же самый флаг, т. е. сделать его единицей, таким образом:

```
 1011101010001100
 0000000000010000   OR
─────────────────
 1011101010011100
             ↑
```

Ниже приводятся несколько приемов использования операции **AND**.

Вы могли заметить, что значение строчной буквы в коде ASCII отличается от значения прописной в точности на 32 (в десятичной системе счисления). Мы можем создать определение, которое осуществляло бы перевод строчного символа в прописной:

```
: ПРОПИСНОЙ   ( строчный-символ -- прописной-символ )
   32 - ;
```

Итак,

```
97 EMIT a ok
97 ПРОПИСНОЙ EMIT A ok
```

К сожалению, данный вариант слова ПРОПИСНОЙ не будет действовать в том случае, если переводимый символ уже является прописным, поскольку весь процесс перевода сводится к простому вычитанию.

Но код ASCII разработан очень мудро. Число 32 выбрано не случайно, а с учетом его представления в двоичной системе счисления. Посмотрите, как выглядят представления прописной и строчной букв A:

```
A    1000001
a    1100001
```

Они отличаются только одним битом, который и представляет число 32. Если мы сбросим этот бит в 0, то независимо от того, была ли буква прописной или строчной, она станет прописной:

```
: ПРОПИСНОЙ   ( строчный-символ -- прописной-символ )
   95 AND ;
```

Число 95 является десятичным эквивалентом двоичного числа 1011111
и совпадает с маской для двоичного представления числа 32. Следовательно,

```
97 ПРОПИСНОЙ EMIT A ok
65 ПРОПИСНОЙ EMIT A ok
```

(Однако поведение рассматриваемого варианта слова ПРОПИСНОЙ оказывается несколько странным по отношению к небуквенным символам. Попытайтесь, к примеру, перевести цифры.)

Слово **XOR** также предназначено для работы с битами. Как отмечалось в гл. 4, при выполнении этой операции истина получается только тогда, когда один из аргументов (но не оба сразу) истинен. Сравним результат выполнения операций **XOR** и **OR**:

```
1000100100001001              1000100100001001
0000001111001000  OR          0000001111001000  XOR
----------------              ----------------
1000101111001001              1000101011000001
```

Если вы применяете операцию **XOR** с аргументом, все биты которого равны единице, то тем самым инвертируете биты второго аргумента.

```
1111111111111111
1000100100001001  XOR
----------------
0111011011110110
```

Таким образом, выражение

```
-1   XOR
```

является двоичной маской или шаблоном инвертирования. (Существует математический термин *дополнение числа до единицы.*)

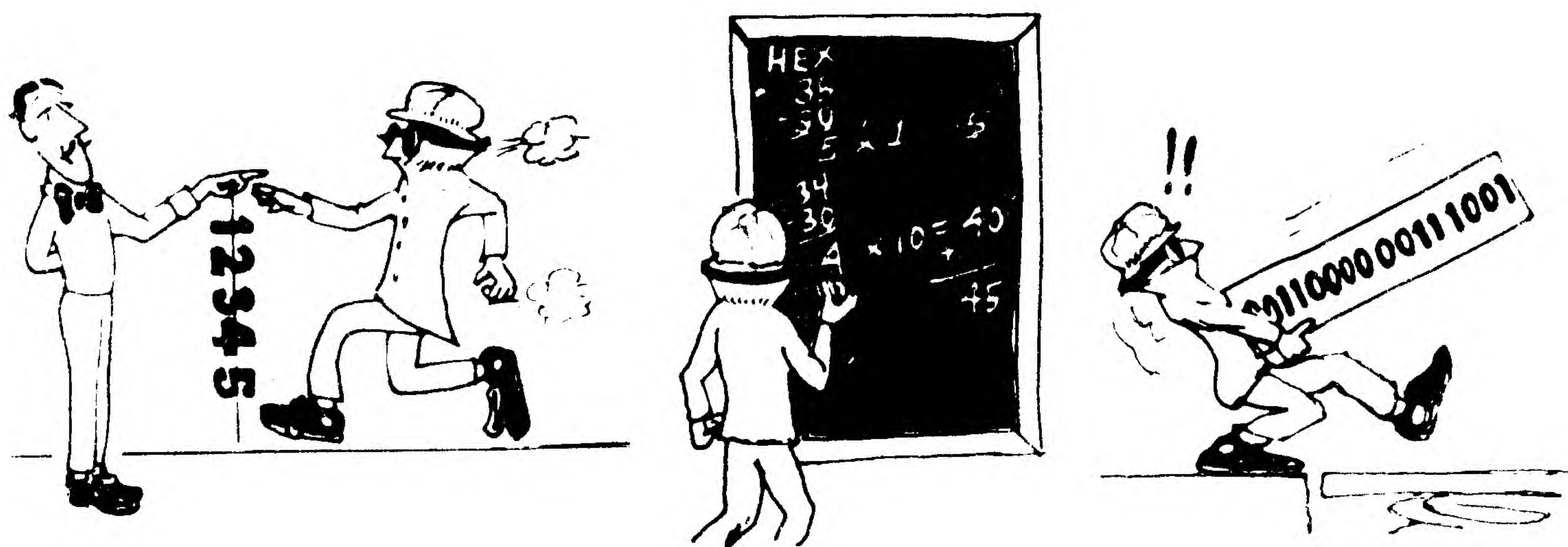
Стандарт-83 и операция NOT. Стандарт-83 изменил первоначальный смысл операции **NOT**. В системах, разработанных до принятия этого Стандарта, слово **NOT** заменяло значение логического аргумента оператора **IF** противоположным, т. е. не нуль (истина) становился нулем (ложью). Оно было синонимом слова

0= , созданным для улучшения читабельности программы. В Стандарте-83 слово **NOT** эквивалентно выражению "—1 XOR" и не сработает в том случае, если значение исходного флага «истина» не представлено как — 1.

Обязательно убедитесь в том, что инвертируемое значение является логическим, а не арифметическим. Выражение "0= NOT" вырабатывает из ненулевого значения правильное логическое значение «истина».

ЧИСЛА СО ЗНАКОМ И БЕЗ ЗНАКА

В первой главе мы ввели слово **NUMBER** (ЧИСЛО).



Если слово **INTERPRET** (ИНТЕРПРЕТАТОР) не может найти введенную строку символов в словаре, то оно передает ее слову **NUMBER**, после чего **NUMBER** пытается прочесть всю совокупность символов как двоичное число. Когда **NUMBER** это удается, прочитанное число помещается в двоичной форме в стек. **NUMBER** не проверяет числа на принадлежность их какому-либо диапазону[1], поэтому может представлять вводимые числа либо как числа со знаком, либо как числа без знака. Например, при вводе любого числа в диапазоне от 32768 до 65535 **NUMBER** представит его в виде числа без знака, а любого значения в диапазоне от —32768 до —1 — как целое в двоичном дополнительном коде. Это важный момент: стек может быть использован для хранения целых чисел со знаком или целых чисел без знака. Будет ли некоторое двоичное значение интерпретироваться как целое со знаком или как целое без знака, зависит от выполняемых над ним операций. Вы выбираете то, что вам больше подходит в данной ситуации, а затем твердо придерживаетесь выбранного варианта.

---

[1] *Для начинающих.* **NUMBER** не проверяет, выходит ли введенное вами в качестве числа одинарной длины значение за рамки соответствующего диапазона. Если вы ввели слишком большое число, то **NUMBER** преобразует его, но сохранит только 16 последних значащих цифр.

Ранее мы ввели слово ., которое выводит на печать из стека значение в виде целого *со знаком:*

```
65535 . -1 ok
```

Слово U. печатает то же самое двоичное представление как число *без знака*:

```
65535 U. 65535 ok
```

U.	(u --)	Вывод числа одинарной длины без знака с одним пробелом после него.

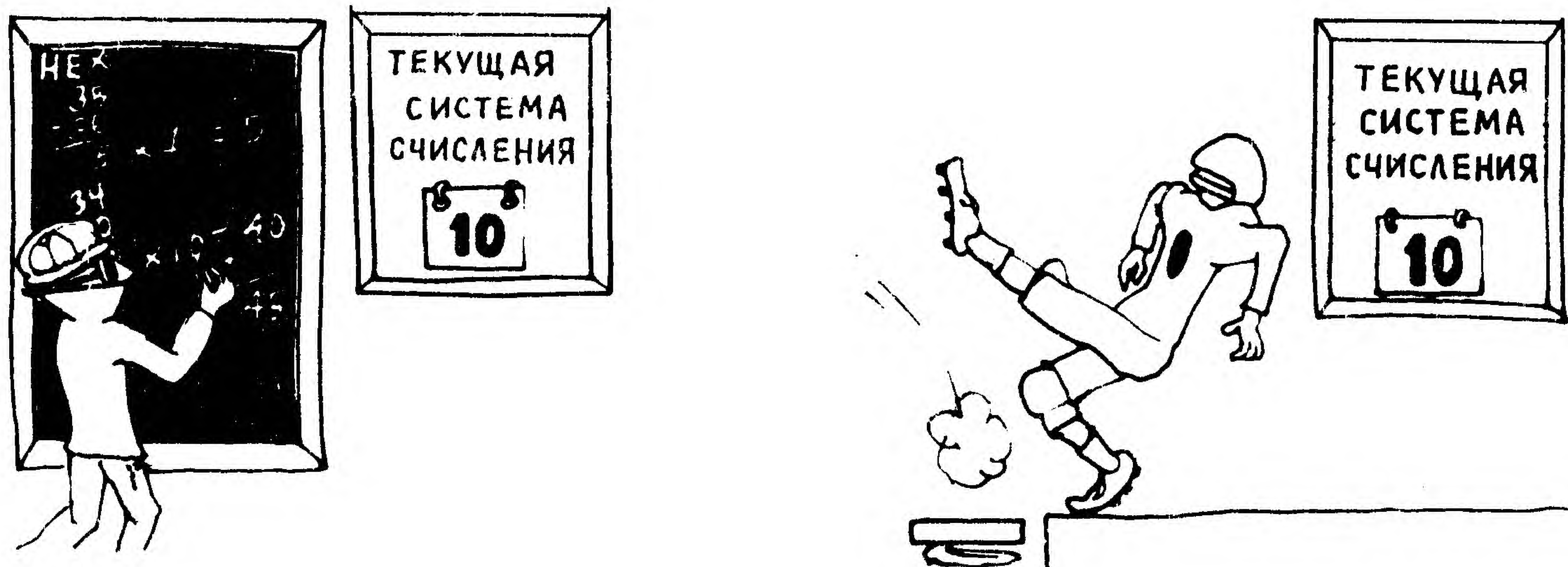
Напоминаем, что буквой n обозначаются числа одинарной длины *со знаком,* а буквой u — числа одинарной длины *без знака.*

Ниже приводятся еще два слова, использующие числа без знака:

U.R	(u ширина --)	Вывод числа без знака. Число выровнено по правой границе поля заданной ширины.
U<	(u1 u2 -- ?)	Помещение на стек истины в том случае, если u1<u2. Оба аргумента рассматриваются как числа одинарной длины без знака.

СИСТЕМЫ СЧИСЛЕНИЯ

После загрузки Форт-системы все преобразования чисел как для ввода, так и для вывода осуществляются в десятичной системе счисления.



Применив перечисленные ниже команды, вы можете сменить текущую систему счисления:

HEX (--) — устанавливает шестнадцатиричную систему счисления;

OCTAL (--) — устанавливает восьмиричную систему счисления (применяется в некоторых системах);

DECIMAL (--) — возвращает десятичную систему.

Вновь принятая система счисления остается таковой до следующего изменения, так что не забудьте объявить **DECIMAL**, как только закончите работать с другой системой счисления.

Рассмотренные команды упрощают преобразования чисел при работе в режиме калькулятора. Если требуется, к примеру, перевести число 100 в шестнадцатиричную систему, вы должны ввести следующее:

```
DECIMAL 100 HEX . 64 ok
```

Для того чтобы перевести шестнадцатиричное число F в десятичную систему (помните, что вы уже имеете дело с шестнадцатиричной системой), нужно ввести:

```
0F DECIMAL . 15 ok
```

Возьмите себе в привычку с этого момента предварять каждое шестнадцатиричное значение нулем, например:

```
0A  0B  0FF
```

что позволит отличать их от системных слов, таких, как **B** в словаре РЕДАКТОРА (EDITOR).

Полезный прием. Определение двоичной (BINARY) или любой другой системы счисления. Начинающие, которые хотят посмотреть, как выглядят числа в двоичной системе, могут ввести следующее определение:

```
: BINARY   2 BASE ! ;
```

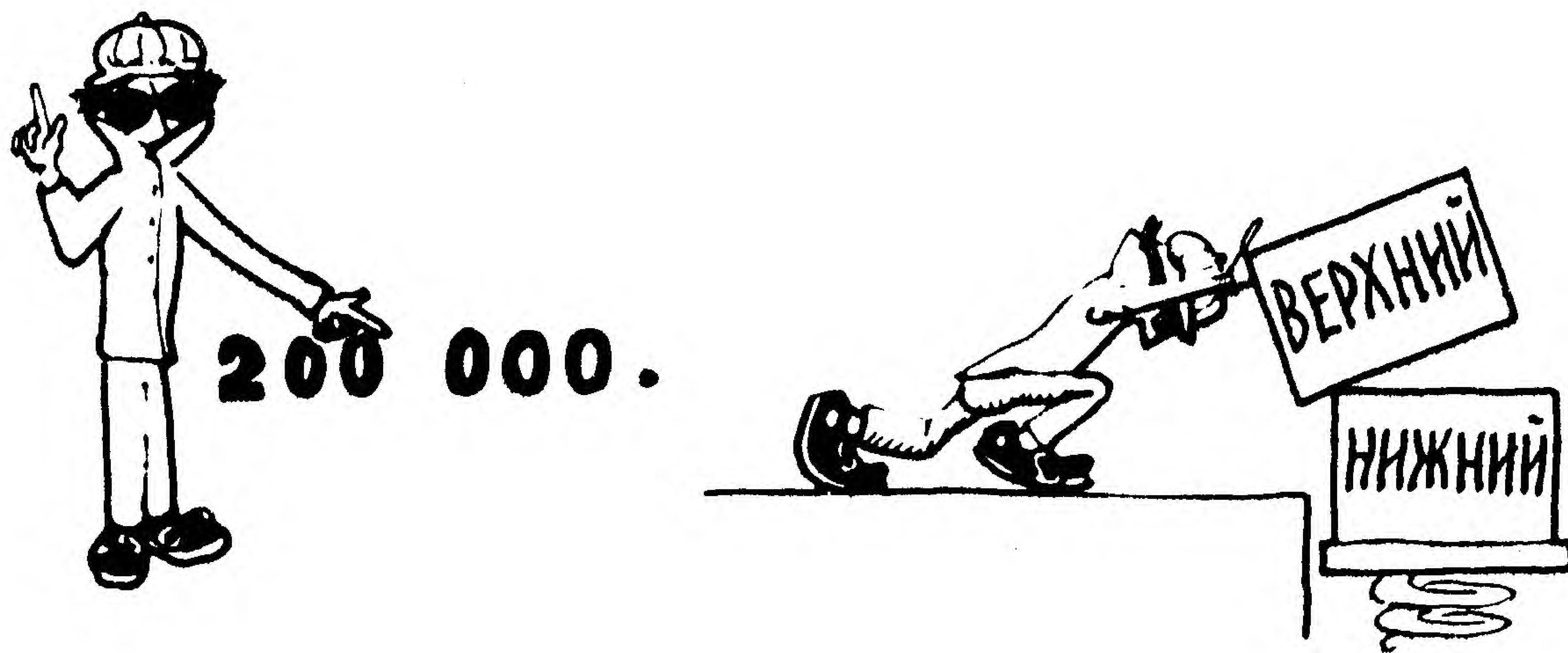
Новое слово BINARY выполняется так же, как и **OCTAL** или **HEX**, но изменяет текущую систему счисления на двоичную. В тех системах, где нет слова **OCTAL**, в порядке эксперимента можно определить:

```
: OCTAL   8 BASE ! ;
```

ЧИСЛА ДВОЙНОЙ ДЛИНЫ

Числа двойной длины составляют диапазон ±2 147 483 647. Большинство Форт-систем до некоторой степени поддерживают работу с числами двойной длины. Для того чтобы вводимое вами (с клавиатуры или из блока) число воспринималось в стеке как число двойной длины, проще всего включить в состав этого числа десятичную точку. Например, когда вы вводите:

```
200000.<return>
```



NUMBER воспринимает запятую как признак того, что данное значение должно быть представлено числом двойной длины и помещает это значение в стек в виде двух последовательных ячеек (ячейкой в Форте называется структура из 16 битов), причем старшая по порядку ячейка помещается над младшей[1].

Слово **D.** выводит число двойной длины без всяких знаков пунктуации:

```
D.            ( d -- )     Печать числа двойной длины со знаком.
```

Здесь d — целое число двойной длины со знаком. Например, если вы введете число двойной длины, а затем выполните операцию **D.**, то компьютер вам ответит:

```
D. 200000 ok
```

В некоторых Форт-системах для выделения дробной части применяются еще четыре знака пунктуации:

```
,   /   -   :
```

В таких системах все перечисленные ниже числа переводятся в одно и то же представление:

```
12345. D. 12345 ok
123.45 D. 12345 ok
1-2345 D. 12345 ok
1/23/45 D. 12345 ok
1:23:45 D. 12345 ok
```

---

[1] *Для специалистов.* В большинстве Форт-систем положение десятичной точки запоминается в некоторой переменной, и вы можете это использовать в своих целях. Более подробную информацию вы найдете в документации по вашей системе. Мы продолжим обсуждение данного вопроса в гл. 10 (см. «Преобразование чисел при вводе»).

Кроме того, в некоторых системах, где применяются арифметические сопроцессоры, на расширенное представление целого числа указывает не десятичная точка, а символ X, например:

```
123456789X
```

а число —12 345 — нет, потому что оно будет представлено как отрицательное число одинарной длины. (Это единственный случай, когда дефис интерпретируется как минус, а не как знак пунктуации.)

Далее мы покажем, как вы можете определить свой собственный эквивалент операции **D.**, чтобы выводить вместе с числом любой знак пунктуации.

ФОРМИРОВАНИЕ ЧИСЕЛ ДВОЙНОЙ ДЛИНЫ БЕЗ ЗНАКА

Приведенные ниже слова:

```
$200.00    12/31/86    372-8493    6:32:59    98.6
```

иллюстрируют типы форматов вывода, которые вы можете создать, определив ваши собственные слова «форматного вывода» Форта. Рассмотрим этот вопрос подробнее.

Самое простое определение форматного вывода вы можете написать следующим образом:

```
: UD.   ( ud -- )   <#  #S  #>   TYPE ;
```

UD. предназначено для вывода числа двойной длины без знака. Слова <# и #> (СКОБКА-ЧИСЛО и ЧИСЛО-СКОБКА) означают начало и конец процесса преобразования числа. В данном определении весь перевод осуществляется единственным словом #S (ЧИСЛА). #S преобразует значение из стека в символы кода ASCII. По этой команде формируется столько цифр, сколько их необходимо для представления числа: незначащие нули она не выводит. Однако всегда выводится по крайней мере одна цифра: если значение равно нулю, то выводится нуль, например:

```
12,345 UD. 12345ok
12. UD. 12ok
0. UD. 0ok
```

Слово **TYPE** (ПЕЧАТЬ) выводит символы, которые составляют число. Заметьте, что между числом и приглашением ok нет пробела. Для того чтобы вывести пробел, вы должны просто добавить слово **SPASE**, как это сделано в приведенном ниже примере:

```
: UD.   ( ud -- )   <#  #S  #>  TYPE SPACE ;
```

Предположим, что у вас в стеке имеется номер телефона, выраженный 32-разрядным целым числом, скажем 372-8493 (помните, что дефис указывает **NUMBER** на то, что число нужно воспринимать как значение двойной длины. В вашей системе это может быть точка). Вы хотите определить некоторое слово, которое будет представлять такое число снова в виде телефонного номера. Назовем его .ТЕЛЕФОН (для вывода номера телефона) и запишем следующее определение:

```
: .ТЕЛЕФОН  ( ud -- )
    <#  # # # # 45 HOLD  #S   #>  TYPE SPACE ;
```

Ваше определение .ТЕЛЕФОН содержит все компоненты слова **UD.** и некоторые другие. Слово Форта #(ЧИСЛО) выводит только одну цифру. Определение форматного вывода числа берет цифры выводимого числа в обратном порядке, поэтому выражение «# # # #» выводит четыре крайние правые цифры номера телефона. Теперь самое время вставить дефис. В таблице значений кода ASCII (см. часть 1 — для начинающих) дефис закодирован десятичным числом 45. Слово Форта **HOLD** (ЗАНЕСТИ) возьмет этот код ASCII и вставит его в строку символов форматируемого числа.

Если в вашей системе имеется слово **ASCII**, то вы можете заменить в приведенном выше определении непонятное выражение "45 HOLD" на более читабельное:

```
ASCII - HOLD
```

Слово **ASCII** (которое мы ввели в первой части данной главы) помещает в стек числовое значение следующего символа из входного потока, в нашем случае — дефис.

Итак, у вас остались три левые цифры. Вы можете воспользоваться выражением «# # #», но проще применить слово **#S**, которое автоматически приведет остаток вашего числа к требуемому виду.

Последовательность <#...#> называется выражением *форматного вывода числа*, поскольку формирует шаблон (справа налево), по которому число должно быть сформатировано.



Теперь представим число двойной длины без знака как календарную дату: 7/15/86. Определение будет выглядеть так:

```
: .ДАТА  ( ud -- )
   <#  # # ASCII / HOLD  # # ASCII / HOLD  # #  #>
   TYPE SPACE ;
```

Проследим за тем, как выполняется приведенное выше определение, но будем помнить, что операции обработки чисел в нем расположены в порядке, обратном выводу этих чисел. Выражение

```
# #  ASCII / HOLD
```

(вы можете вместо "ASCII/" использовать 47) выводит две крайние правые цифры (год) и крайний слэш. При следующем вхождении этого выражения выводятся две средние цифры (день) и левый слэш. Наконец, "# #" выводит две крайние цифры (месяц).

Вы можете определить:

```
: /nn   ( ud -- ud)   # # ASCII / HOLD ;
:.ДАТА  ( ud -- )     <#  /nn /nn  # #  #> TYPE SPACE;
```

Поскольку вы сами управляете процессом преобразования чисел, можно на самом деле переводить различные числа из одной системы счисления в другую. Это может пригодиться вам при форматном выводе времени (часов и минут). Например, у вас в стеке время записано в секундах, а вы хотите определить слово, которое выражало бы это время в часах, минутах и секундах. Определение в данном случае может выглядеть следующим образом[1]:

```
: SEXTAL   6 BASE ! ;
: :00  ( Ud -- Ud) #  SEXTAL # DECIMAL ASCII : HOLD ;
: СЕКУНДЫ   <#  :00 :00 #S  #>  TYPE SPACE ;
```

Для форматного вывода секунд и минут вы используйте слово :00. Как секунды, так и минуты вычисляются по модулю 60, значит, правой цифрой может быть любая цифра до девяти, а левой — цифра от нуля до пяти включительно. Поэтому в своем определении :00 вы преобразуете первую цифру (она является правой) как десятичное число, затем переходите по слову SEXTAL в шестиричную систему (с основанием 6) и преобразуете левую цифру, после чего возвращаетесь в десятичную систему и вставляете символ двоеточия. После того как слово :00 преобразует секунды и минуты, # S переведет оставшиеся часы. Так, если у вас в стеке время задано как 4500 с, то в результате вы получите:

```
4500. .СЕКУНДЫ 1:15:00 ok
```

---

[1] *Для начинающих.* См. полезный прием, описанный на с. 164.

(Если продолжительность дня измерять в секундах, то 86400 с -- это слишком много для 16-разрядного числа.)

В табл. 7.2 сведены слова Форта, использующиеся при форматизации чисел. (Обратите внимание на условные обозначения в конце таблицы, которые напоминают вам о смысле символов "n", "d" и т. д.)

Таблица 7.2

Форматирование чисел

<#	Начало процесса преобразования числа. В стеке должно находиться *число двойной длины без знака*
#	Преобразование одной цифры и помещение ее в выходную символьную строку. # доставляет цифру в ЛЮБОМ СЛУЧАЕ — если вы подали этому слову на вход неверное цифровое значение, то и в этом случае вы получите нуль для каждого #
#S	Преобразование числа (цифры за цифрой) до тех пор, пока в результате не получится нуль. Всегда доставляется *по крайней мере одна цифра* (нуль, если число равно нулю)
c HOLD	Вставка в форматируемую символьную строку на текущую позицию символа, значение которого в коде ASCII находится в стеке
n SIGN	Вставка знака «–» в выходную строку в том случае, если третье число в стеке отрицательное (это число из стека выбирается — см. сноску в следующем разделе)
#>	Завершение преобразования числа и помещение в вершине стека счетчика символов и адреса (именно эти аргументы требуются для TYPE)

ВЫРАЖЕНИЕ	СОСТОЯНИЕ СТЕКА	ТИП АРГУМЕНТОВ
<# ... #>	(d -- a u) или	32-разрядный без знака
	(u 0 -- a u)	16-разрядный без знака
<# ... n SIGN #>	(:d: -- a u)	32-разрядный со знаком, где :d: является абсолютным значением d, а n - верхней ячейкой d
	или	
	(:n: 0 -- a u)	16-разрядный со знаком, где :n: - абсолютное значение n

Условные обозначения:

n, n1 ... - 16-разрядные числа со знаком;
a - адрес;
d, d1 ... - 32-разрядные числа со знаком;
u, u1 ... - 16-разрядные числа без знака;
c - значение символа в коде ASCII.

ФОРМАТИРОВАНИЕ ЧИСЕЛ ОДИНАРНОЙ ДЛИНЫ СО ЗНАКОМ

До сих пор мы форматировали только числа двойной длины без знака. Хотя структура <#...#> применима именно к таким числам, попробуем воспользоваться ею и для других типов чисел, выполнив определенные манипуляции со стеком. Например, рассмотрим простейший вариант системного определения **D.** (оно выводит число двойной длины *со знаком*):

```
: D.  ( d -- )
   DUP >R  DABS  <# #S  R> SIGN  #>  TYPE SPACE ;
```

Слово **SIGN**, которое должно располагаться внутри выражения форматного вывода, вставляет знак "—" в строку символов лишь в том случае, если верхний символ в стеке является отрицательным. Следовательно, мы должны сохранить копию верхней ячейки (содержащей знак) в стеке возвратов для дальнейшего использования.

Так как <# требует наличия лишь чисел двойной длины без знака, мы должны взять абсолютное значение нашего числа двойной длины *со знаком* с помощью слова **DABS**. Теперь расположение аргументов в стеке соответствует выражению форматного вывода. Затем #S осуществляет перевод цифр справа налево, после чего мы заносим в стек знак. Если этот знак отрицательный, то **SIGN** добавляет к форматированной строке минус[1]. Так как нам нужно, чтобы знак минус располагался слева, включаем **SIGN** справа в структуру <#...#>. В некоторых случаях, например в бухгалтерских расчетах, может потребоваться вывод отрицательных чисел в виде 12345—. В подобной ситуации мы должны поместить слово **SIGN** *слева* в выражение <#...#>, как показано ниже:

```
<# (SIGN #S  #>
```

---

[1] *Для пользователей более ранних систем.* Слово **SIGN** имеет свою историю. Первоначально в качестве аргумента этого слова выступал третий элемент стека. Таким образом, для того чтобы определить слово **D.**, нужно было писать

```
: D.  ( d)
   SWAP OVER DABS  <#  #S SIGN  #>  TYPE SPACE ;
```

Выражение "SWAP OVER" помещает копию верхней ячейки (той, что со знаком) на дно стека. Чтобы упростить выражение форматного вывода, операцию **ROT** решено было перенести в определение слова **SIGN**.

Данное соглашение действовало в системах фиг-Форт и системах полиФорт, разработанных до введения Стандарта-83. В системах Форт-79 и Форт-83, а также Форт-системах MVP и MMS используется соглашение о передаче знака через вершину стека, что в большей степени соответствует механизму передачи данных через стек.

Если вы располагаете старой версией, определите **SIGN** следующим образом:

```
: SIGN  ( n)  0< IF  ASCII - HOLD  THEN ;
```

Определим слово, которое выводило бы число двойной длины со знаком с десятичной точкой и двумя десятичными значениями после точки. Поскольку такая форма часто используется для представления какой-либо суммы в долларах и центах, назовем это слово .$ и определим его следующим образом:

```
: .S  ( d -- )
   DUP >R  DABS  <#  # #  ASCII . HOLD  #S
   R> SIGN  ASCII S HOLD  #> TYPE SPACE ;
```

Проверим его:

```
2000.00 .$ $2000.00
```

или даже так:

```
2,000.00 .$ S2000.00
```

Рекомендуем вам сохранить определение .$, которое пригодится нам в дальнейшем в некоторых примерах.

Вы можете также описать форматы чисел одинарной длины. Например, если вы хотите использовать число одинарной длины без знака, то просто поместите в стеке перед словом <# нуль. Такое число одинарной длины легко превратить в число двойной длины, которое настолько мало, что его часть, содержащаяся в верхней по порядку ячейке, равна нулю.

Для представления числа одинарной длины со знаком вам достаточно поместить нуль в старшую по порядку ячейку. Но вы также должны сохранить копию числа со знаком для **SIGN** и, кроме того, оставить абсолютное значение этого числа во втором элементе стека. Все необходимые действия выполняет следующее выражение:

```
( n -- )   DUP >R  ABS 0  <#  #S  R> SIGN  #>
```

Ниже приводятся «стандартные» выражения, которые применяются для вывода различных видов чисел:

ВЫВОДИМОЕ ЧИСЛО	СЛОВУ <# ПРЕДШЕСТВУЕТ ВЫРАЖЕНИЕ
32-разрядное без знака	(ничего)
31-разрядное плюс знак	DUP >R DABS (чтобы сохранить знак в третьем элементе стека для SIGN)
16-разрядное без знака	0 (чтобы получить фиктивную, старшую по порядку, часть)
15-разрядное плюс знак	DUP >R ABS 0 (чтобы сохранить знак)

ОПЕРАЦИИ НАД ЧИСЛАМИ ДВОЙНОЙ ДЛИНЫ

Слова для выполнения операций над числами двойной длины имеются не во всех Форт-системах. В некоторые системы они включены на правах выборочных, т. е. прежде чем ими пользоваться, вы должны их загрузить.

Ниже приводится перечень слов двойной длины для выполнения математических операций.

D+	(d1 d2 -- d-сумма)	Сложение двух 32-разрядных чисел.
D-	(d1 d2 -- d-разность)	Вычитание одного 32-разрядного числа из другого (d1-d2).
DNEGATE	(d -- -d)	Изменение знака 32-разрядного числа на противоположный.
DABS	(d1 -- :d:)	Занесение в стек абсолютного значения 32-разрядного числа.
DMAX	(d1 d2 -- d-max)	Занесение в стек максимального из двух 32-разрядных чисел.
DMIN	(d1 d2 -- d-min)	Занесение в стек минимального из двух 32-разрядных чисел.
D=	(d1 d2 -- ?)	Занесение в стек истины в случае равенства d1 и d2.
D0=	(d -- ?)	Занесение в стек истины, если d равно нулю.
D<	(d1 d2 -- ?)	Занесение в стек истины, если d1 меньше d2.
DU<	(ud1 ud2 -- ?)	Занесение в стек истины, если ud1 меньше ud2. Оба числа без знака.
D.R	(d ширина --)	Вывод 32-разрядного числа со знаком. Число выравнивается справа внутри поля заданной ширины.

Буква D в начале каждого выражения означает, что указанная операция может выполняться только над числами двойной длины, а цифра 2 в начале слова, в частности **2SWAP** и **2DUP**, — что данная операция может выполняться как над числами двойной длины, так и над парами чисел одинарной длины. Пример выполнения операции **D+**:

```
200000. 300000. D+ D. 500000 ok
```

В следующей таблице перечислены часто используемые слова Форта, которые выполняются над комбинацией чисел одинарной и двойной длины[1]:

UM*	(u1 u2 -- ud)	Перемножение двух 16-разрядных чисел. Все значения без знака. (В предыдущих версиях данная операция называлась U*.)
UM/MOD	(ud u1 -- u2 u3)	Деление 32-разрядного числа на 16-разрядное. В стек заносятся 16-разрядные остаток и частное (частное в вершину). Частное округлено до ближайшего меньшего целого[1]. Все значения без знака. (В предыдущих версиях данная операция называлась U/MOD).
M*	(n1 n2 -- d-произведение)	Перемножение двух 16-разрядных чисел. Все значения со знаком.
M+	(d n -- d-сумма)	Сложение 32-разрядного и 16-разрядного чисел. Результат 32-разрядный.
M/	(d n -- n-частное)	Деление 32-разрядного числа на 16-разрядное. Результат 16-разрядный. Все значения со знаком.
M*/	(d n u -- d)	Умножение 32-разрядного на 16-разрядное и деление промежуточного результата тройной длины на 16-разрядное число (d*n/u). Результат 32-разрядный.

Слово **UM*** является командой быстрого умножения, где под результат отводится 32 разряда. Эта команда может быть использована как базовая при определении других операций умножения. Аналогичным образом все команды деления могут быть определены через базовую операцию **UM/MOD**.

Ниже приводится пример выполнения операции **M+**:

```
200,000 7 M+ D. 200007 ok
```

С помощью операции **M*/** мы можем переопределить наш прежний вариант слова % так, что оно будет выполняться с аргументом двойной длины:

```
: %  ( d n% -- d)  100 M*/ ;
```

[1] См. сноску в разд. «Операции деления» (гл. 2).

Например:

```
200.50 15 % D. 3007 ok
```

Если вы загрузили определение **.$** (которое мы приводили ранее), то можете ввести следующий текст:

```
200.50 15 % .$ $30.07
```

Можно переопределить данное выше определение R% таким образом, чтобы получать округленный результат двойной длины

```
: R%  ( d n% -- d)  10 M*/  5 M+  1 10 M*/ ;
```

и тогда

```
200.50 15 R% .$ $30.08
```

Заметим, что **M***/ является единственным «штатным» словом Форта, которое осуществляет умножение аргумента двойной длины. Для того чтобы умножить, скажем, 200 000 на 3, мы должны подставить единицу в качестве фиктивного делителя:

```
200,000 3 1 M*/ D. 600000 ok
```

так как 3/1 эквивалентно 3.

M*/ также является единственным словом Форта, которое осуществляет деление с результатом двойной длины. Так, при делении 200 000 на 4 мы должны подставить единицу в качестве фиктивного числителя:

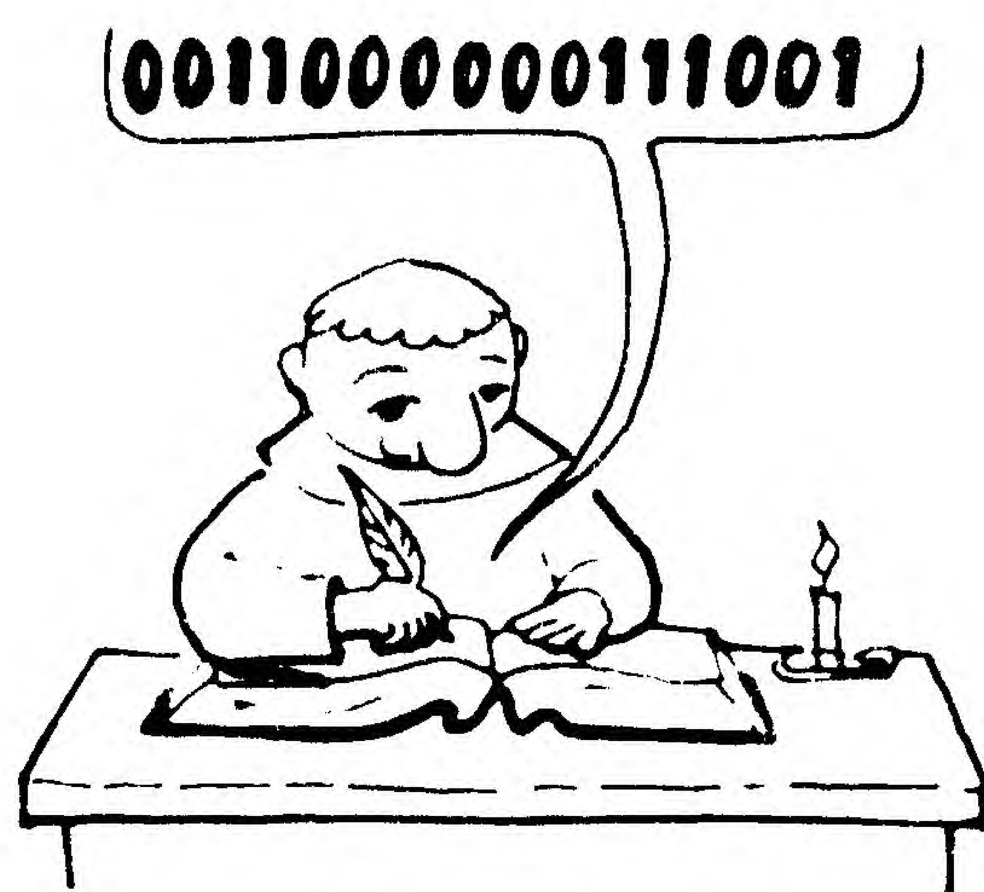
```
200,000 1 4 M*/ D. 50000 ok
```

ИСПОЛЬЗОВАНИЕ ЧИСЕЛ В ОПРЕДЕЛЕНИЯХ

Если в определении содержится число, например,

```
: БОЛЬШЕ-НА-ДВАДЦАТЬ  ( n -- n+20)  20 + ;
```

то это число компилируется в словарь в двоичной форме точно также, как оно выглядит в стеке.



Двоичное значение числа зависит от системы счисления, которая существует в системе на момент *компиляции* данного определения. Например, при вводе

```
HEX   : БОЛЬШЕ-НА-ДВАДЦАТЬ  ( n -- n+20)   14 + ;   DECIMAL
```

словарное определение должно содержать шестнадцатиричное значение 14, что соответствует десятичному значению 20 (16 + 4). Впредь слово НЕСКОЛЬКО-БОЛЬШЕ будет всегда добавлять к содержимому стека эквивалент десятичного числа 20, независимо от текущей системы счисления. В том случае, когда вы поместите слово **HEX** *внутрь* определения, основание системы счисления будет изменяться при *выполнении* данного определения. Например, если вы определяете:

```
DECIMAL
: ПРИМЕР   HEX 20 .  DECIMAL ;
```

то число компилируется как двоичный эквивалент десятичного числа 20, поскольку во время компиляции текущей была **DECIMAL** (десятичная) система счисления. *Во время выполнения* произойдет следующее:

```
ПРИМЕР 14 ok
```

Наше число выдается в шестнадцатиричной системе.

Заметьте, что число, помещенное внутри некоторого определения, называется *литералом*. (В отличие от слов, присутствующих в этом определении, которые ссылаются на другие определения, значение числа заключено в самом числе.)

Ниже приводится перечень слов Форта, рассмотренных в настоящей главе.

U. (u --) Вывод числа одинарной длины без знака с одним пробелом после него.

U.R (u ширина --) Вывод числа без знака. Число выровнено по правой границе поля заданной ширины.

U< (u1 u2 -- ?) Помещение на стек истины в том случае, если u1<u2. Оба аргумента рассматриваются как числа одинарной длины без знака.

Стековая нотация для форматирования чисел:

ВЫРАЖЕНИЕ	СОСТОЯНИЕ СТЕКА	ТИП АРГУМЕНТОВ
<# ... #>	(d -- a u) или (u 0 -- a u)	32-разрядный без знака 16-разрядный без знака
<# ... n SIGN #>	(:d: -- a u) или (:n: 0 -- a u)	32-разрядный со знаком, где :d: является абсолютным значением d, а n - верхней ячейкой d 16-разрядный со знаком, где :n: - абсолютное значение n

---

Операции над числами двойной длины:

D. (d --) Печать числа двойной длины со знаком.

D+ (d1 d2 -- d-сумма) Сложение двух 32-разрядных чисел.

D- (d1 d2 -- d-разность) Вычитание одного 32-разрядного числа из другого (d1-d2).

DNEGATE (d -- -d) Изменение знака 32-разрядного числа на противоположный.

DABS (d1 d2 -- |d|) Занесение в стек абсолютного значения 32-разрядного числа.

DMAX (d1 d2 -- d-max) Занесение в стек максимального из двух 32-разрядных чисел.

DMIN (d1 d2 -- d-min) Занесение в стек минимального из двух 32-разрядных чисел.

D= (d1 d2 -- ?) Занесение в стек истины в случае равенства d1 и d2.

D0= (d -- ?) Занесение в стек истины, если d равно нулю.

D< (d1 d2 -- ?) Занесение в стек истины, если d1 меньше d2.

DU<	(ud1 ud2 -- ?)	Занесение в стек истины, если ud1 меньше ud2. Оба числа без знака.
D.R	(d ширина --)	Вывод 32-разрядного числа со знаком. Число выравнивается справа внутри поля заданной ширины.

Смешанные операции:

UM*	(u1 u2 -- ud)	Перемножение двух 16-разрядных чисел. Все значения без знака.(В предыдущих версиях данная операция называлась U*.)
UM/MOD	(ud u1 -- u2 u3)	Деление 32-разрядного числа на 16-разрядное. В стек заносятся 16-разрядные остаток и частное (частное в вершину). Частное округлено до ближайшего меньшего целого. Все значения без знака. (В предыдущих версиях данная операция называлась U/MOD.)
M*	(n1 n2 -- d-произведение)	Перемножение двух 16-разрядных чисел. Все значения со знаком.
M+	(d n -- d-сумма)	Сложение 32-разрядного и 16-разрядного чисел. Результат 32-разрядный.
M/	(d n -- n-частное)	Деление 32-разрядного числа на 16-разрядное. Результат 16-разрядный. Все значения со знаком.
M*/	(d n u -- d)	Умножение 32-разрядного на 16-разрядное и деление промежуточного результата тройной длины на 16-разрядное число (d*n/u). Результат 32-разрядный.

Условные обозначения:

n, n1 ... – 16-разрядные числа со знаком;
a – адрес;
d, d1 ... – 32-разрядные числа со знаком;
u, u1 ... – 16-разрядные числа без знака;
c – значение символа в коде ASCII.

ОСНОВНЫЕ ТЕРМИНЫ

Арифметический сдвиг влево и вправо. Процесс сдвига всех разрядов числа, за исключением знакового, влево или вправо для эффективного выполнения операций деления и умножения на степень числа 2 с сохранением знака.

ASCII. Стандартизованная система представления вводимых или выводимых символов в виде побайтных значений. Полное

название — стандартный американский код для обмена информацией.

Байт. Стандартный термин для обозначения 8-разрядного значения.

Восьмиричная система. Система счисления по основанию 8.

Двоичная система. Система счисления по основанию 2.

Десятичная система. Система счисления по основанию 10.

Дополнение. Число, равное исходному по абсолютной величине, но с противоположным знаком. Для того чтобы вычислить разность 10 — 4, компьютер сначала определит дополнение 4 (т. е. —4), а затем произведет сложение 10 + (—4).

Знаковый разряд. Разряд, который для чисел со знаком указывает, является ли данное число положительным или отрицательным, а для чисел без знака является старшим значащим разрядом.

Литерал. Число или символ, значение которого заключено в нем самом; в Форте — это число, появляющееся внутри определения.

Маска. Значение, которое может быть «наложено» на другое значение, в результате чего определенные разряды маскируются, а высвечиваются только интересующие нас разряды.

Слово. В Форте — это определенный элемент словаря, во всех других случаях — это термин для обозначения 16-разрядного значения.

Форматизация чисел. Процесс представления двоичного числа выводимыми на дисплей символами, как правило, в специальной форме, например: 3/13/81 или $47.93.

Число без знака. Число, которое по определению положительное.

Число одинарной длины без знака. Целое, значение которого может лежать в диапазоне от 0 до 65535.

Шестнадцатиричная система. Система счисления по основанию 16.

Ячейка. Термин Форта для 16-разрядного значения.

УПРАЖНЕНИЯ

ДЛЯ НАЧИНАЮЩИХ

7.1. Вероника Вэйнрайт не могла вспомнить верхнее значение для чисел одинарной длины со знаком. У нее не было книги, по которой она могла бы справиться, а была только Форт-система, поэтому она написала определение N-MAX с использованием цикла **BEGIN ... UNTIL**. После выполнения этого определения у нее получилось следующее:

```
32767 ok
```

Восстановите написанное ею определение.

7.2. (Данное упражнение позволит вам приобрести навыки работы с битами.) Прежде всего, если вы не сделали этого ранее, определите слово BINARY.

а) Разряды внутри 16-разрядной ячейки нумеруются с 0 до 15 справа налево, так что нулевым разрядом является самый младший бит, а 15-м — самый старший.

Определите слово с именем БИТ, которое переводило бы номер разряда (от 0 до 15) в маску, соответствующую этому разряду, например 0 бит соответствует маске 1, бит 1 — маске 2, бит 3 — маске 4 и т. д. (*Совет*: проще всего использовать цикл **DO**.)

б) Пусть в стеке находится значение, представляющее некоторый массив из 16 разрядов (назовем его «битовый1»). Определите слово с именем УСТАНОВИТЬ-БИТ, которое устанавливало бы в единицу заданный бит в массиве «битовый1» при следующей стековой нотации: (битовый1 номер-бита — битовый2). Например, если в стеке находится битовый шаблон, равный двоичному числу 1000, и вы выполняете последовательность команд 1 УСТАНОВИТЬ-БИТ (в десятичной системе), то получите в результате 1010.

в) Определите слово ОЧИСТИТЬ-БИТ, которое сбрасывает заданный бит. Стековая нотация та же, что и для слова УСТАНОВИТЬ-БИТ. Например:

```
BINARY 111111111

DECIMAL  5 ОЧИСТИТЬ-БИТ  7 ОЧИСТИТЬ-БИТ
BINARY U. 101011111
```

г) Определите слово ДАЙ-БИТ, вносящее в стек указанный бит (выбранный посредством маски из массива «битовый»), который мог бы служить аргументом оператору **IF** (т. е. если бит установлен в единицу, то в стек помещается значение «истина», если нет, — «ложь»). Стековая нотация выглядит следующим образом: (битовый номер-бита — бит). Например:

```
BINARY 1001  DECIMAL
DUP  0 ДАЙ-БИТ . 1 ok
DUP  1 ДАЙ-БИТ . 0 ok
3 ДАЙ-БИТ . 8 ok
```

д) Определите слово с именем ПЕРЕКЛЮЧИТЬ-БИТ с той же стековой нотацией, что и для слов УСТАНОВИТЬ-БИТ и ОЧИСТИТЬ-БИТ, которое переключало бы заданный бит (устанавливало бы его в единицу, если бы он был сброшен, и наоборот).

е) Пусть заданы две битовые последовательности, причем вторая получена путем изменения значений нескольких битов первой. Определите слово с именем ИЗМЕНЕНИЕ, которое воспроизводило бы битовую маску с измененными битами.

7.3. Напишите определение, которое заставило бы зазвонить колокольчик на вашем терминале три раза. Убедитесь в том, что перерыв в звучании достаточен для того, чтобы отдельные звонки не слились в один длинный звонок. Всякий раз при звучании колокольчика на экране терминала должно появляться слово БИП.

Упражнения 7.4 и 7.5 посвящены выполнению операций над числами двойной длины.

7.4 а) Перепишите созданные в гл. 5 определения перевода значений температур из одной шкалы в другую в предположении, что вводимые в результирующие значения температур должны быть представлены целыми числами двойной длины со знаком с коэффициентом масштабирования 10 (т. е. должны быть умножены на 10). Например, если вы вводите фактическое значение 10.5°, то оно будет представлено 32-разрядным целым числом со значением 105.

б) Напишите слово для форматного вывода с именем .ГРАДУСЫ, которое будет выдавать 32-разрядное целое с коэффициентом масштабирования 10 как строку цифр с десятичной точкой и одной дробной цифрой. Например:

```
12.3 .ГРАДУСЫ<return> 12.3 ok
```

в) Преобразуйте следующие значения температур:

0.0 °F в °C
212.0 °F в °C
20.5 °F в °C
16.0 °C в °F
—40.0 °C в °F
100.0 °K в °C
100.0 °K в °F
233.0 °K в °C
233.0 °K в °F

7.5. а) Напишите программу, которая вычисляет значение полинома
$7x^2 + 20x + 5$
при заданном x и выводит результат в виде числа двойной длины.

б) Какое максимальное значение может принимать x, чтобы при вычислении результата как 32-разрядного числа со знаком не произошло переполнения?

ДЛЯ ВСЕХ

7.6. Неопытный пользователь экспериментирует с шестнадцатиричными числами и пытается вернуться снова к десятичной форме, вводя команду DEC. Несмотря на то что Форт-система выводит ok, пользователю кажется, что перевод из одной системы счисления в другую не происходит. Что случилось?

7.7. Напишите слово, которое бы выводило числа от 0 до 16 (десятичных) в десятичной, шестнадцатиричной и двоичной системах счисления в три столбца, т. е.

```
DECIMAL  0    HEX  0    BINARY     0
DECIMAL  1    HEX  1    BINARY     1
DECIMAL  2    HEX  2    BINARY    10
       . . .
DECIMAL 16    HEX 10    BINARY 10000
```

7.8. Введите число 3.7 и дважды выполните команду . (точка). Объясните, что вы получили и почему. Введите число 65536. и дважды выполните команду . . Объясните результат. Попытайтесь ввести число 65538. (с десятичной точкой).

7.9. Если вы вводите

```
..<return>
```

(две точки, *не* разделенные пробелом) и система отвечает вам ok, что это означает?

7.10. Напишите определение для форматного вывода номера телефона, которое одновременно выводит через слэш номер района *тогда и только тогда, когда* в этот номер включен номер района, т. е.

```
555-1234 .ТЕЛЕФОН 555-1234 ok
213/372-8493 .ТЕЛЕФОН 213/372-8493 ok
```

(В некоторых системах вместо слэша и дефиса может быть выведена десятичная точка.)

Глава 8

ПЕРЕМЕННЫЕ, КОНСТАНТЫ И МАССИВЫ

Из материала семи предыдущих глав вы уже поняли, что программирующие на Форте используют стек для передачи аргументов от одного слова к другому. Если программистам требуется хранить числа более продолжительное время, они применяют переменные и константы. В этой главе будет показано, как Форт-система трактует переменные и константы и каким образом можно непосредственно получить доступ к участку памяти.

ПЕРЕМЕННЫЕ (ОБЩИЕ СВЕДЕНИЯ)

Для начала приведем пример ситуации, когда вам понадобилось бы воспользоваться переменной, например для хранения даты[1]. В первую очередь создадим переменную (VARIABLE) с именем ДАТА[2]:

```
VARIABLE ДАТА
```

Если сегодня 12-е число, то мы вводим: 12 ДАТА !, т. е. помещаем 12 в стек, затем указываем имя переменной и, наконец, выполняем слово ! (ЗАПОМНИ). Это выражение записывает число 12 в переменную ДАТА. Для обратного действия нужно ввести:

---

[1] *Для начинающих.* Предположим, ваш компьютер выдает банковские счета на протяжении дня, и каждый такой счет должен включать дату. Вам не хотелось бы хранить дату все время в стеке и, кроме того, дата не должна быть частью определения, иначе его пришлось бы переопределять каждый день. Вам нужна переменная.

[2] *Для пользователей систем фиг-Форт.* Чтобы вариант **VARIABLE** вашей системы совпадал с описанным в книге, введите определение
: VARIABLE 0 VARIABLE ;

ДАТА @, иными словами, назвать переменную, а затем выполнить слово @ (ВЫБРАТЬ). Указанное выражение выбирает число 12 и помещает его в стек. Таким образом, выражение

```
ДАТА @ . 12 ок
```

выведет на печать дату.

На Форте это делается еще проще с помощью слова, определение которого приводится ниже:

```
: ?  @ . ;
```

Поэтому вместо «ДАТА-выборка-точка» мы можем просто набрать:

```
ДАТА ? 12 ok
```

ДАТА будет иметь значение 12 до тех пор, пока вы не измените его. Для того чтобы изменить значение переменной, необходимо записать в нее новое число.

```
13 ДАТА ! ok
ДАТА ? 13 ок
```

Понятно, что можно определить и дополнительные переменные для месяца и года:

```
VARIABLE ДАТА   VARIABLE МЕСЯЦ   VARIABLE ГОД
```

а затем слово с именем !ДАТА (для «запоминания-даты»), например:

```
: !ДАТА  ( месяц день год -- )  ГОД !  ДАТА !  МЕСЯЦ ! ;
```

и использовать его следующим образом:

```
7 31 88 !ДАТА ок
```

После этого нужно определить слово с именем .ДАТА (для «вывода-даты»):

```
: .ДАТА   МЕСЯЦ ?  ДАТА ?  ГОД ? ;
```

Ваша Форт-система уже имеет ряд определенных переменных, одна из которых — **BASE**. Она содержит основание текущей системы счисления. На самом деле определения систем счисления

HEX, DECIMAL (и **OCTAL**, если в вашей системе таковая имеется) весьма просты:

```
: DECIMAL   10 BASE ! ;
: HEX   16 BASE ! ;
: OCTAL   8 BASE ! ;
```

Вы можете работать с любой системой счисления, просто загрузив ее основание в **BASE**[1]:

Если где-нибудь в определении системных слов, осуществляющих преобразования входных и выходных чисел, вы найдете выражение **BASE @**, значит, текущее значение **BASE** используется в процессе такого преобразования. Следовательно, одна и та же программа может работать с числами в любой системе счисления. Отсюда мы можем сделать следующее формальное заключение об использовании переменных: в Форте уместно создавать переменные для тех значений, используемых внутри определения, которые могут быть изменены в любой момент после того, как это определение уже скомпилировано.

БОЛЕЕ ПОДРОБНО О ПЕРЕМЕННЫХ

Создавая некоторую переменную, например ДАТА, с помощью выражения VARIABLE ДАТА, вы фактически компилируете новое слово с именем ДАТА в словарь. Упрощенно это выглядит так[2]:

ДАТА
Командный код, соответствующий переменным
Память под фактическое значение, которое будет сюда записано

---

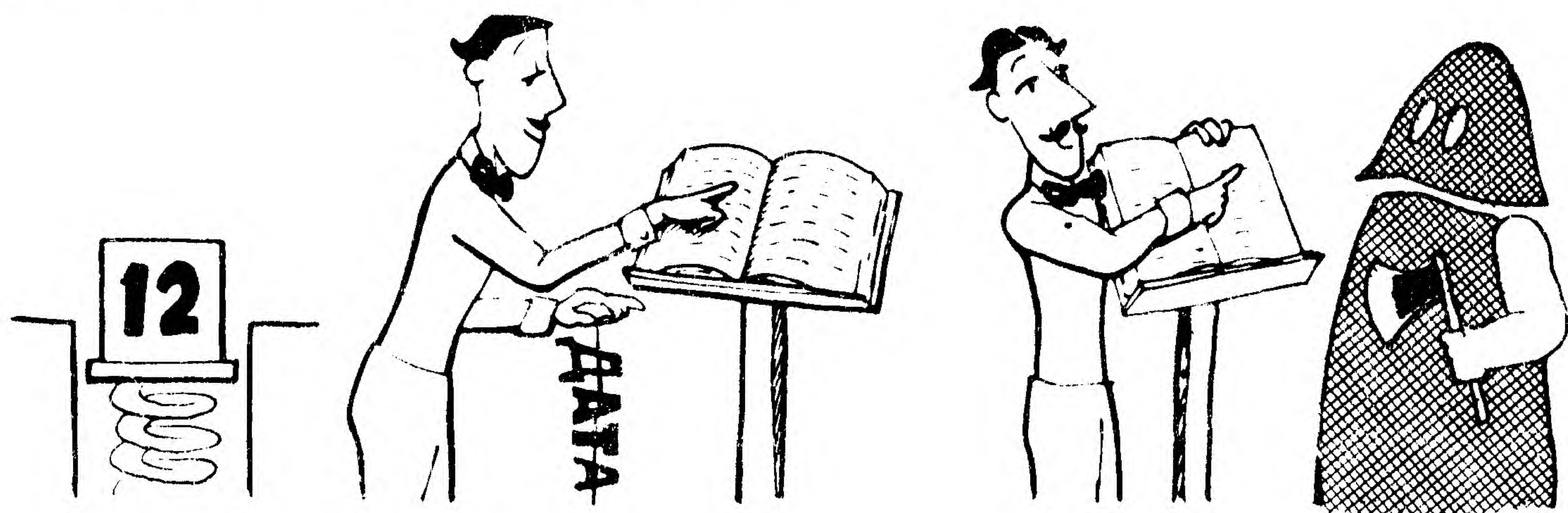
[1] *Для специалистов.* Трехбуквенный код, например, имя терминала некоего аэропорта, можно запомнить как число одинарной длины без знака в системе счисления с основанием 36. Например:

```
: АЛЬФА   36 BASE ! ;
АЛЬФА ok
ZAP U. ZAP ok
```

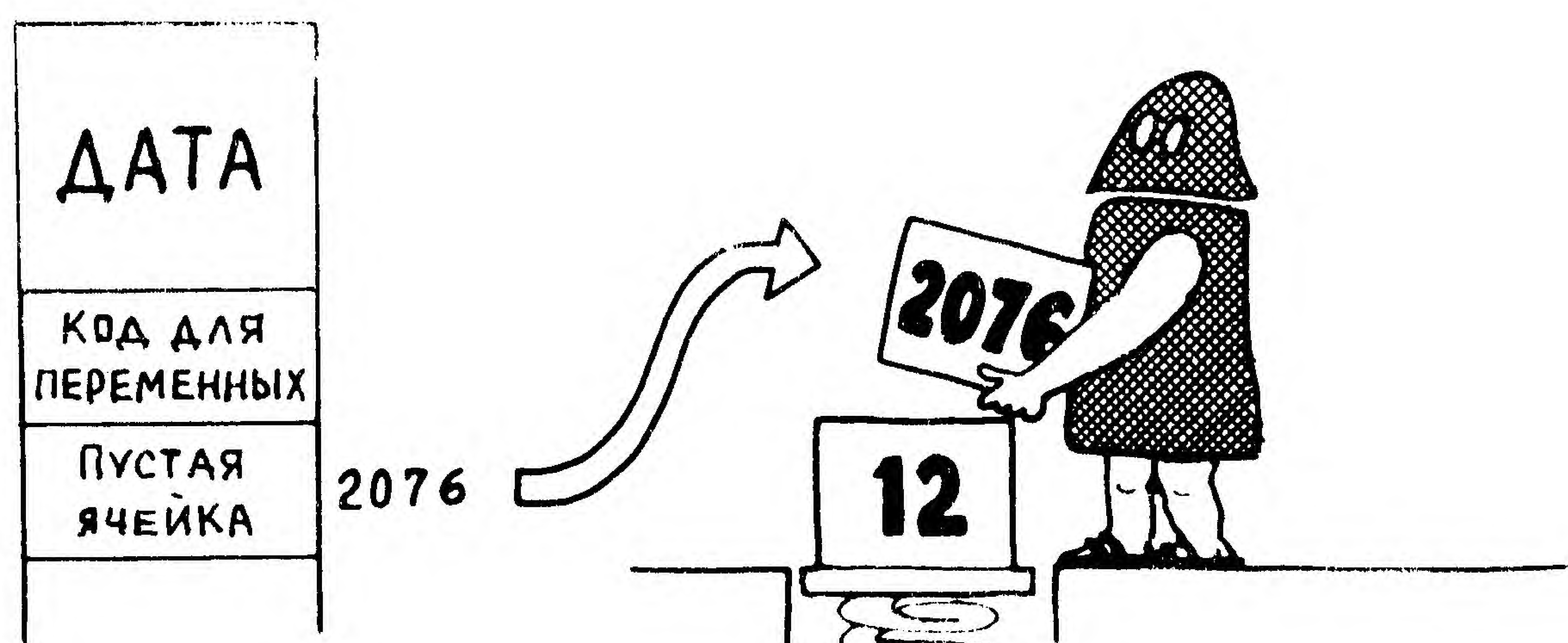
[2] *Для специалистов.* Как на самом деле выглядит в памяти элемент словаря, мы покажем в следующей главе.

Слово ДАТА аналогично любому другому слову в вашем словаре, за исключением того, что оно определено с помощью слова **VARIABLE**, а не :, поэтому вы не должны специфицировать функции своего определения. Само имя слова **VARIABLE** предопределяет, что должно произойти. А происходит следующее.

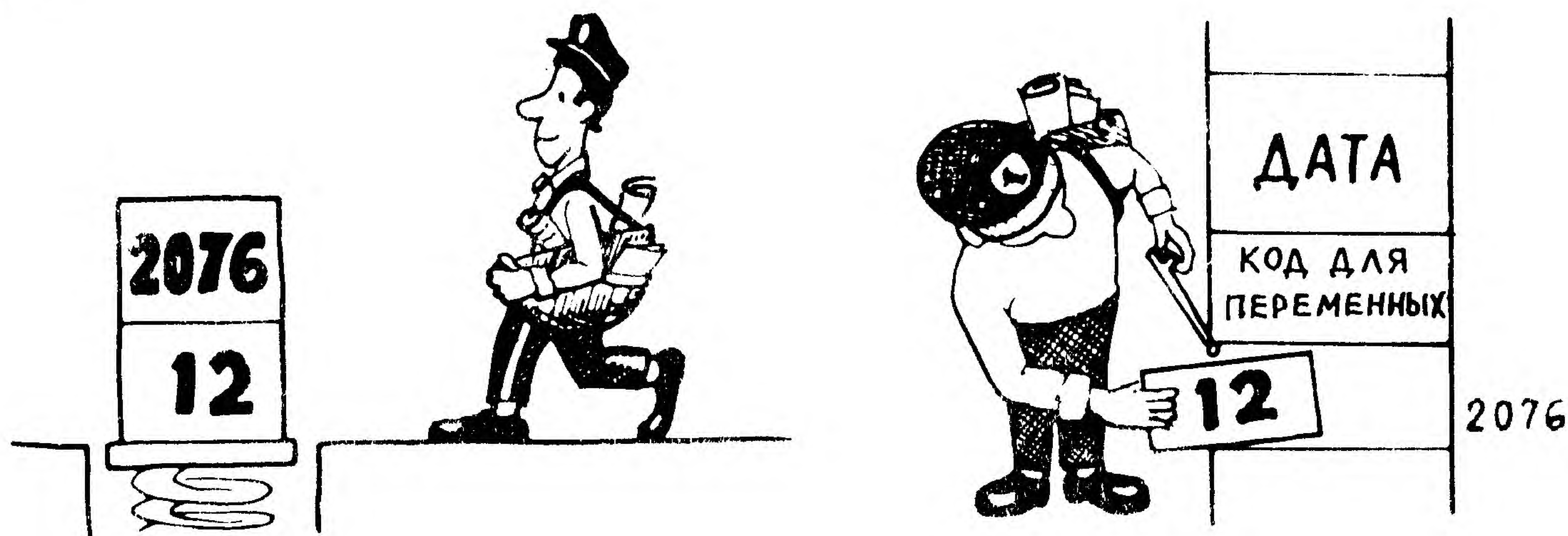
Когда вы вводите 12 ДАТА !



число 12 поступает в стек, после чего интерпретатор текста ищет слово ДАТА в словаре и, найдя его, передает на исполнение **EXECUTE**.



EXECUTE выполняет переменную путем копирования адреса «пустой» ячейки этой переменной (куда будет послано значение) в стек[1]:



[1] *Для начинающих.* В программировании адресом называется число, которое определяет участок машинной памяти. Например, по адресу 2076 (адреса, как правило, выражаются шестнадцатиричными числами без знака) может содержаться 16-разрядное представление значения 12. Здесь 2076 — адрес, 12 — содержимое.

Слово! выбирает адрес (из вершины) и значение (под ним) и запоминает это значение по выбранному адресу. Новое число замещает любое другое число, находившееся прежде по данному адресу.

(Чтобы запомнить порядок аргументов, можно провести такую аналогию: думайте о том, что сначала нужно положить посылку, а уже потом сверху приклеить адресную бирку.)

Для слова @ требуется только один аргумент: адрес, который в данном случае обеспечивается именем переменной, например ДАТА@.



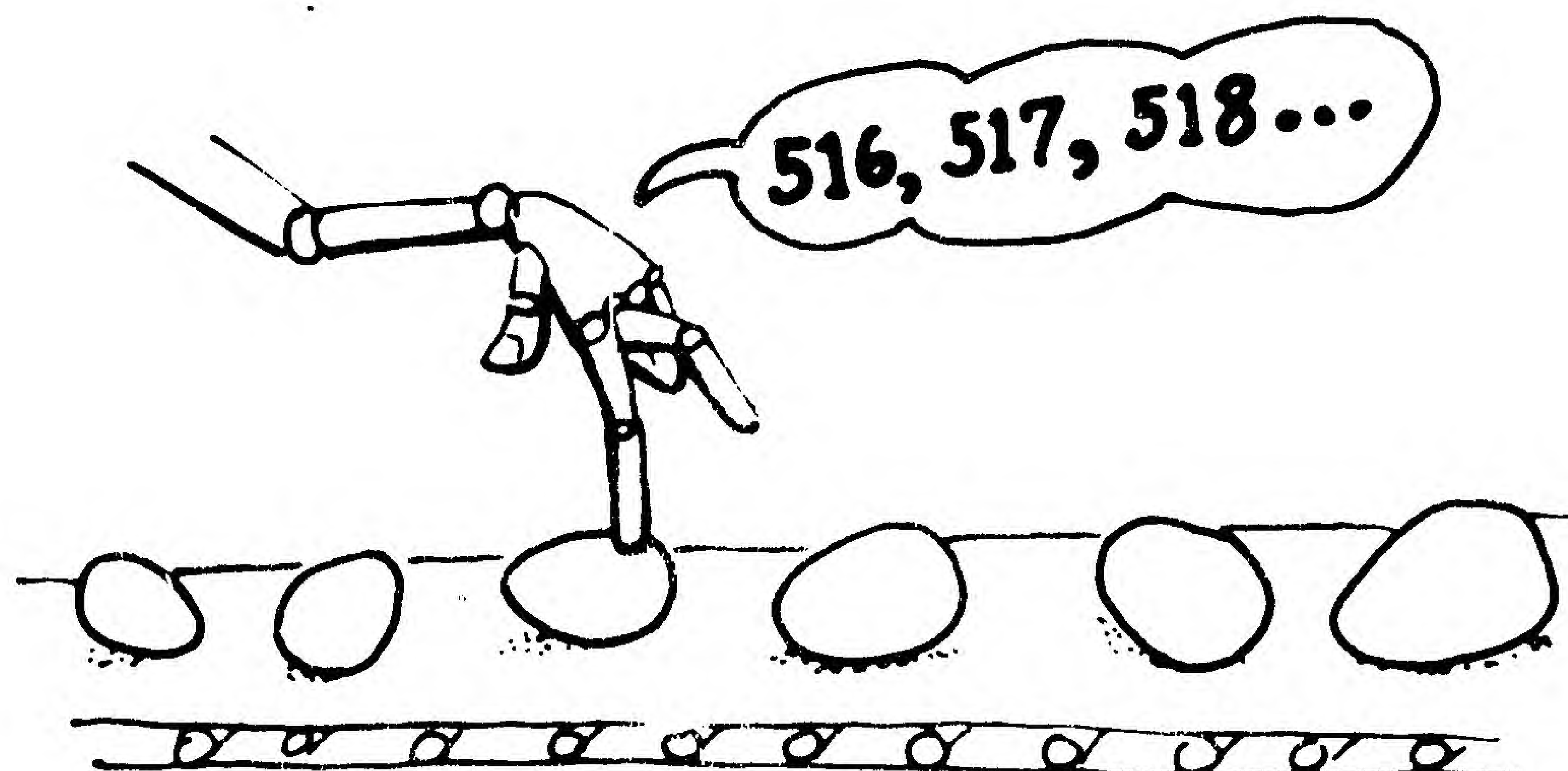
Используя выбранное из стека значение в качестве адреса, слово @ помещает содержимое, находящееся по данному адресу, в стек, предварительно сняв с последнего адрес. (Содержимое участка памяти остается прежним.)

ПЕРЕМЕННЫЕ В КАЧЕСТВЕ СЧЕТЧИКА

В Форте переменная представляет собой идеальное средство для счетчика. Вернемся к примеру с машиной для упаковки яиц и предположим, что нам нужна информация о том, сколько яиц проходит по ленте конвейера за один день. (Этот пример вы должны выполнить за своим терминалом, так что по ходу изложения набирайте на клавиатуре текст и вводите его.)

Сначала мы определяем

```
VARIABLE ЯЙЦА
```



в которой будем вести подсчет. Каждое утро мы будем начинать подсчет с нуля, поэтому нам придется загружать в переменную ЯЙЦА нуль, используя слово, определение которого выглядит так:

```
: УСТАНОВИТЬ   0 ЯЙЦА ! ;
```

После этого где-нибудь в нашей программе по упаковке яиц нужно определить слово, которое всякий раз, когда яйцо минует электрический «глазок» на конвейере, выполняет следующее выражение:

```
1 ЯЙЦА +!
```

Слово +![1] добавляет заданное значение к содержимому (любому) по данному адресу. Таким образом, выражение 1 ЯЙЦА + ! увеличивает счетчик яиц на единицу. Для иллюстрации изложенного поместим это выражение внутрь некоторого определения:

```
: ЯЙЦО    1 ЯЙЦА +! ;
```

А в конце дня выясним, сколько яиц прошло через конвейер, набрав на клавиатуре ЯЙЦА?. Теперь проверим:

```
УСТАНОВИТЬ ok
ЯЙЦО ok
ЯЙЦО ok
ЯЙЦО ok
ЯЙЦА ? 3 ok
```

Ниже приводится перечень слов, которые мы уже рассмотрели в настоящей главе.

VARIABLE xxx	(--) xxx: (-- a)	Создание переменной с именем xxx. Слово xxx при выполнении помещает в стек свой адрес.
!	(n a --)	Запоминание числа одинарной длины по заданному адресу.
@	(a -- n)	Замещение адреса его содержимым.
?	(a --)	Вывод значения по заданному адресу с последующим пробелом.
+!	(n a --)	Сложение числа одинарной длины с содержимым заданного адреса.

[1] *Для любознательных.* Это слово обычно определяют на уровне языка Ассемблера, определение же на языке высокого уровня имеет вид:

```
: +!  ( приращение a -- )  DUP @  ROT +  SWAP ! ;
```

КОНСТАНТЫ

Если в переменных обычно хранятся значения, которые могут изменяться, то константы используются для хранения значений, которые изменению *не подлежат*. На Форте мы создаем константу и тут же устанавливаем ее значение, например:

```
220 CONSTANT ПРЕДЕЛ
```

Здесь определена константа с именем ПРЕДЕЛ, которой присвоено значение 220. Теперь мы имеем право подставлять слово ПРЕДЕЛ вместо значения

ПРЕДЕЛЫ
Командный код, соответствующий константам
220

```
: ?ЖАРКО  ( температура -- )
   ПРЕДЕЛ > IF ." Опасно -- Уменьшите нагрев! " THEN ;
```

В том случае, когда число в стеке больше 220, выдается предупреждающее сообщение. Заметьте, что, говоря ПРЕДЕЛ, мы получаем *значение*, а не адрес. Нам здесь не требуется «выборка». В этом и состоит основное отличие переменных от констант. Дело в том, что при работе с переменной нам нужен адрес, чтобы иметь возможность как выборки значения, так и его запоминания. При использовании же константы всегда требуется значение (мы никогда в нее ничего не запоминаем).

Одним из примеров применения констант может служить именование аппаратного адреса. Допустим, что программа для управления фотокамерой с помощью микропроцессора содержит следующее определение:

```
: СНИМОК  ЗАТВОР ОТКРЫТЬ  ВРЕМЯ ВЫДЕРЖАТЬ  ЗАТВОР ЗАКРЫТЬ ;
```

Здесь слово ЗАТВОР определено как константа при условии, что его выполнение обеспечивает аппаратный адрес затвора фотокамеры. Оно может быть определено так:

```
HEX
3E27 CONSTANT ЗАТВОР
DECIMAL
```

Слова ОТКРЫТЬ и ЗАКРЫТЬ могут быть просто определены:

```
: ОТКРЫТЬ   ( а -- )   1 SWAP ! ;
: ЗАКРЫТЬ   ( а -- )   0 SWAP ! ;
```

так что выражение ЗАТВОР ОТКРЫТЬ запишет единицу по адресу затвора, и он откроется.

Использование в определениях констант, а не изображений самих чисел является важным элементом хорошего стиля программирования. Прежде всего наличие констант делает вашу программу более читабельной. Все определения Форта должны быть так же самодокументированы, как определение СНИМОК.

Не менее существенно и то, что значения могут изменяться (могут изменяться, к примеру, аппаратные средства). Если в такой ситуации вам достаточно внести изменения в один фрагмент программы — в определение константы, — то все остальное будет сделано автоматически, без вероятных пропусков, как это имело бы место при корректировке вручную.

Третье преимущество заключается в том, что в компилируемой форме определение, содержащее константу, занимает меньший объем памяти, чем то же определение, но с изображением числа вместо константы. Если некоторое число применяется неоднократно, то получаемый при этом выигрыш перекрывает расходы на описание константы. Поэтому во многих Форт-системах часто повторяющиеся числа определены как константы:

```
0 CONSTANT 0
1 CONSTANT 1
и т.д.
```

В дальнейшем будем считать, что в вашей системе имеются следующие определения констант **FALSE** (ЛОЖЬ) и **TRUE** (ИСТИНА):

```
0 CONSTANT FALSE
-1 CONSTANT TRUE
```

CONSTANT xxx	(n --) xxx: (-- n)	Создание константы с именем xxx и значением n. Слово xxx при своем выполнении заносит в стек n.
FALSE	(-- f)	Занесение в стек логического значения ложь (0).
TRUE	(-- t)	Занесение в стек логического значения истина (-1).

ПЕРЕМЕННЫЕ И КОНСТАНТЫ ДВОЙНОЙ ДЛИНЫ

Вы можете определить переменную двойной длины с помощью слова **2VARIABLE**, например:

```
2VARIABLE ДАТА
```

После этого вы можете использовать слова ФОРТА **2!** (ЗАПОМНИ ДВА) и **2 @** (ВЫБЕРИ ДВА) для доступа к переменной двойной длины. Можно записать число двойной длины в такую переменную, просто записав

```
800000. ДАТА 2!
```

и выбрать значение из нее, введя

```
ДАТА 2@ D. 800000 ok
```

Вы можете также запомнить полностью дату (месяц, число, год) в такой переменной:

```
7/16/86 ДАТА 2!
```

и выбрать их назад:

```
ДАТА 2@  .ДАТА 7/16/81 ok
```

в предположении, что у вас загружен вариант определения .ДАТА, приводимый в последней главе.

Можно определить константу двойной длины с помощью слова **2CONSTANT**, например

```
200000. 2CONSTANT ЯБЛОКИ
```

после чего слово ЯБЛОКИ будет помещать в стек число двойной длины:

```
ЯБЛОКИ D. 200000 ok
```

Судя по префиксу 2, мы можем также использовать слово **2CONSTANT** при определении *пары* чисел одинарной длины. Можно помещать два числа под одним именем просто из соображений удобства, а также для резервирования памяти в словаре.

Вспомните (см. гл. 5), что вы можете с помощью выражения 355 113*/ умножить некоторое число на аппроксимацию *pi*. Слово **2CONSTANT** позволяет запомнить эти два целых:

```
355 113 2CONSTANT PI
```

а впоследствии применить выражение PI */, например:

```
10000 PI */ . 31415 ok
```

Ниже приводится перечень слов применительно к структурам данных двойной длины.

2VARIABLE xxx	(--) xxx: (-- a)	Создание переменной двойной длины с именем xxx. Слово xxx при выполнении помещает на стек свой адрес.
2CONSTANT xxx	(d --) xxx: (-- d)	Создает константу двойной длины с именем xxx и значением d. Слово xxx при выполнении помещает в стек значение d.
2!	(d a --)	Запоминание числа двойной длины по заданному адресу.
2@	(a -- d)	Занесение в стек числа двойной длины, расположенного по заданному адресу.

МАССИВЫ

Как вам уже известно, выражение VARIABLE ДАТА создает определение и выделяет память для значения одинарной длины:

ДАТА
Код
Память для значения одинарной длины

А если необходимо выделить память для 10 или 20 значений одинарной длины под одним и тем же именем? Такая структура называется *массивом*. На форте массив строится следующим образом.

В первую очередь мы создаем массив с помощью непривычного определяющего слова **CREATE** (СОЗДАТЬ)[1]:

```
CREATE МОЙ-МАССИВ
```

[1] *Для пользователей систем фиг-Форта.* В вашей системе имеется слово **CREATE**, но оно отличается от упоминаемого здесь и редко используется. Чтобы вы могли и далее следить за ходом событий, переопределите его следующим образом:

```
: CREATE <BUILD DOES> ;
```

Как и **VARIABLE, CREATE** компилирует в словарь новое имя (вашего массива) вместе с кодами, которые специфицируют его действия. Но при этом память под данные *не выделяется.*

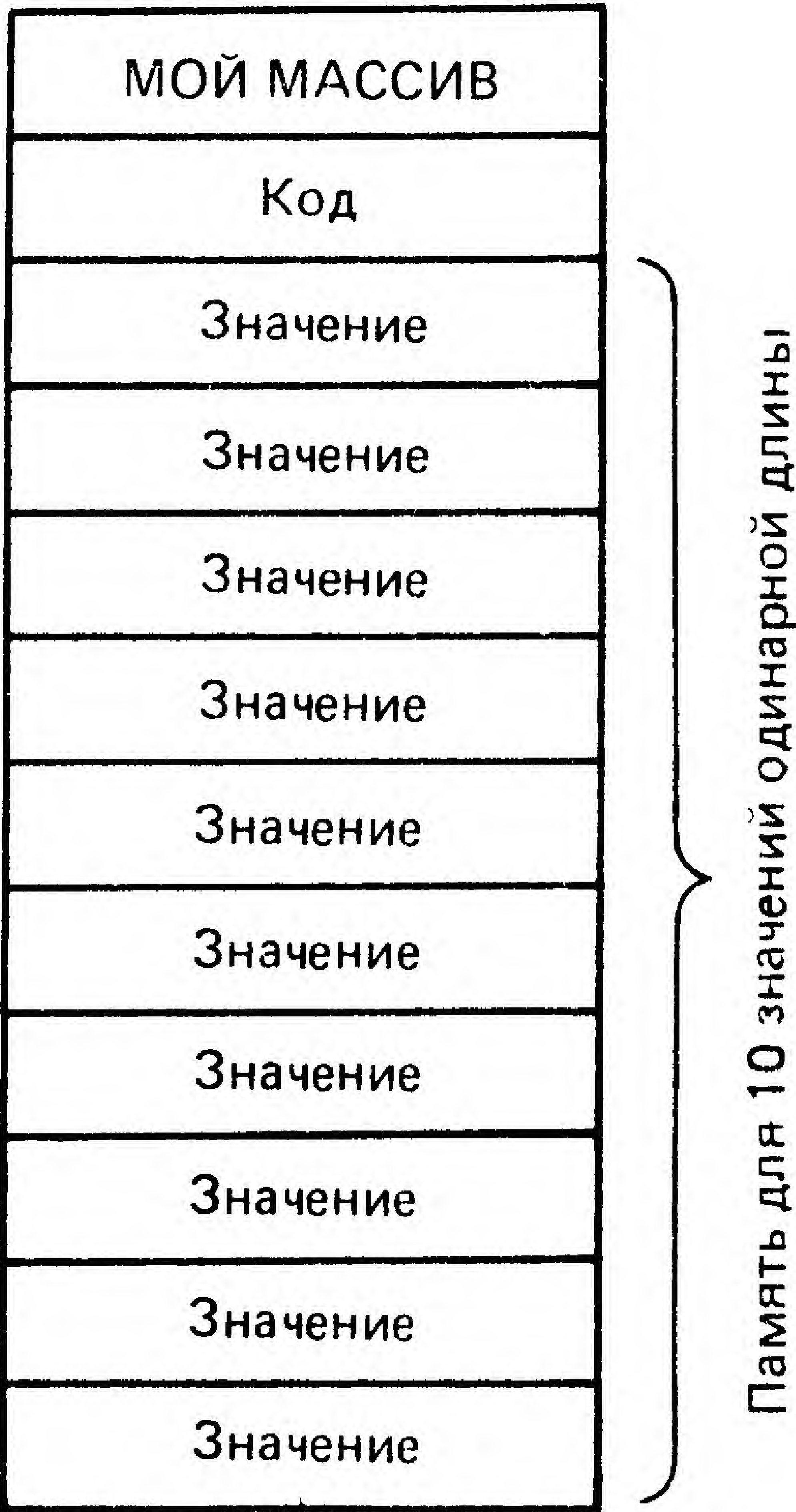
МОЙ МАССИВ
Код

Каким образом мы впоследствии отведем память под созданный массив? Это делается посредством слова **ALLOT** (ВЫДЕЛИТЬ), которое выбирает из стека в качестве аргумента число *байтов*, резервируемое для массива.

Если требуется выделить память под 10 значений одинарной длины, то нужно ввести

```
20 ALLOT
```

(Значение одинарной длины занимает два байта.)



Когда вы исполняете слово, определенное как переменная Форт-система помещает в вершину стека адрес значения. Таким же образом, когда вы исполняете слово, созданное с помощью **CREATE**, в вершину стека заносится адрес начала массива (первого значения).

Проиллюстрируем применение массива на следующем примере. Предположим, что в нашей лаборатории имеется не одна, а *пять* горелок, на которых нагреваются различные жидкости.



Мы можем с помощью слова ?ЖАРКО проверять, не превышает ли температура нагрева каждой из пяти горелок установленного для нее максимального значения, если определим ПРЕДЕЛ не как константу, а как массив. Присвоим этому массиву имя ПРЕДЕЛЫ:

```
CREATE ПРЕДЕЛЫ  10 ALLOT
```

	Адреса ↓
ПРЕДЕЛЫ	
Код	
Память для предельного значения температуры горелки 0	3162
Память для предельного значения температуры горелки 1	3164
Память для предельного значения температуры горелки 2	3166
Память для предельного значения температуры горелки 3	3168
Память для предельного значения температуры горелки 4	316A

Допустим, что мы устанавливаем предельное значение температуры для горелки 0 220°. Запомним это значение посредством следующей фразы:

```
220 ПРЕДЕЛЫ !
```

так как ПРЕДЕЛЫ доставляет адрес первой ячейки нашего массива. Установим предельное значение температуры для горелки

1 равным 340° и запомним его, добавив два байта к адресу исходной ячейки:

```
340 ПРЕДЕЛЫ 2+ !
```



Мы можем запомнить предельные значения для горелок 2, 3 и 4, добавляя к исходному адресу «смещения» 4, 6 и 8. Так как смещение всегда равно удвоенному номеру горелки, определим полезное слово:

```
: ПРЕДЕЛ  ( номер-горелки -- адрес-предельного-значения)
   2*  ПРЕДЕЛЫ + ;
```

чтобы по номеру соответствующей горелки, находящемуся в стеке, вычислять адрес, который отстоит от начала на величину соответствующего смещения[1].

После всех преобразований полезность слова ПРЕДЕЛ возросла в такой степени, что мы можем переопределить слово ?ЖАРКО:

```
: ?ЖАРКО    ( температура номер-горелки -- )
   ПРЕДЕЛ @ >  IF ." Опасно -- Уменьшите нагрев! " THEN ;
```

[1] 1. *Для начинающих.* В этом случае номер горелки называется *индексом* массива. Индекс — это относительный указатель элемента памяти. Умножая индекс на 2, мы получаем *смещение* относительно начала массива. Смещение равно фактическому числу байтов между началом массива и искомым элементом.

2. Мы нумеруем горелки с 0 до 4, вместо того чтобы нумеровать их с 1 до 5, по той причине, что хотим использовать сами номера горелок в качестве индексов. То, что большинство людей называют «первым» в какой-то последовательности, программисты называют «нулевым». При желании вы можете пронумеровать горелки с 1 по 5, но тогда вам придется добавлять в определении слов самого высокого уровня корректирующий фрагмент (простое выражение «1 —»), как это всегда делается.

Примеры выполнения нового варианта этого слова приведены ниже:

```
210 0 ?ЖАРКО ок
230 0 ?ЖАРКО Опасно -- Уменьшите нагрев! ок
300 1 ?ЖАРКО ок
350 1 ?ЖАРКО Опасно -- Уменьшите нагрев! ок
```

и т.д.

CREATE xxx	(--) xxx: (-- a)	Создание заголовка в словаре с именем xxx. Слово xxx при выполнении заносит в стек свой адрес.
ALLOT	(n --)	Резервирование в поле параметров слова, определенного последним, n дополнительных байтов.

ИСПОЛЬЗОВАНИЕ МАССИВА СЧЕТЧИКОВ

Вернемся на нашу птицеферму. Приведем еще один пример на использование массива. Каждый его элемент служит как бы отдельным счетчиком. Следовательно, мы можем дифференцированно вести подсчет числа коробок «очень крупных» яиц, «крупных» яиц и т. д., упакованных машиной.

Вспомните, что в приведенном выше определении РАЗМЕР-ЯИЦ (см. гл. 4) у нас было четыре категории стандартных яиц и две категории нестандартных.

```
0 БРАК
1 МЕЛКИЕ
2 СРЕДНИЕ
3 КРУПНЫЕ
4 ОЧЕНЬ КРУПНЫЕ
5 ОШИБКА
```

Давайте создадим массив из шести ячеек:

```
CREATE СЧЕТЧИКИ  12 ALLOT
```

Счетчики будут увеличиваться с помощью слова +!, так что мы должны иметь средства «обнуления» всех элементов массива перед началом процесса подсчета. Выражение

```
СЧЕТЧИКИ  12 0 FILL
```

заполняет нулями 12 байтов, начиная с адреса СЧЕТЧИКИ. Если в вашей Форт-системе имеется слово **ERASE**, то в данной ситуации лучше воспользоваться им. Это слово заполняет заданное число байтов нулями. Ниже показан пример его использования.

```
СЧЕТЧИКИ 12 ERASE
```

FILL (ЗАПОЛНИТЬ)	(a u b --)	Заполнение n байтов памяти, начиная с заданного адреса, значением b.
ERASE (ОЧИСТИТЬ)	(a u --)	Заполнение n байтов памяти, начиная с заданного адреса, нулями.

Иногда удобно помещать это выражение внутрь определения:

```
: УСТАНОВИТЬ   СЧЕТЧИКИ 12  ERASE ;
```

Далее определим слово, которое по заданному номеру категории яиц (от 0 до 5) даст нам адрес одного из счетчиков, например:

```
: СЧЕТЧИК    ( номер-категории -- a)   2*   СЧЕТЧИКИ + ;
```

и еще одно слово для добавления единицы к счетчику с заданным номером:

```
: УЧЕТ    ( номер-категории -- )   СЧЕТЧИК   1 SWAP   +! ;
```

Здесь 1 служит приращением для слова +!, а **SWAP** располагает его аргументы в требуемом порядке, т. е. (n адрес - -).

Теперь, например, выражение 3 УЧЕТ увеличит значение счетчика, соответствующего категории крупных яиц.

Определим слово, которое переводит вес на дюжину в номер категории[1]:

```
: КАТЕГОРИЯ   ( вес-на-дюжину -- номер-категории)
   DUP   18 < IF   0  ELSE
   DUP   21 < IF   1  ELSE
   DUP   24 < IF   2  ELSE
   DUP   27 < IF   3  ELSE
   DUP   30 < IF   4  ELSE
                   5
   THEN THEN THEN THEN THEN  SWAP DROP ;
```

[1] *Для специалистов.* В конце главы будет приведено более простое определение.

(К тому времени, когда процесс вычисления подойдет к выражению SWAP DROP, в стеке будут находиться два значения: вес, который мы размножили с помощью команды **DUP**, и номер категории, расположенный в вершине. Нам требуется только номер категории. Выражение SWAP DROP убирает вес.)

Например, выражение 25 КАТЕГОРИЯ оставит в стеке число 3. Приведенное выше определение слова КАТЕГОРИЯ напоминает наше прежнее определение РАЗМЕР-ЯИЦ, но, следуя стилю Форта (слова должны создаваться по возможности более короткими), мы убрали из этого определения выдаваемые сообщения и определили еще одно слово, которое по заданному номеру сорта яиц выдает сообщение[1]:

```
: МАРКИРОВКА  ( номер-категории -- )
   DUP  0= IF ." Брак "                ELSE
   DUP 1 = IF ." Мелкие "              ELSE
   DUP 2 = IF ." Средние "             ELSE
   DUP 3 = IF ." Крупные "             ELSE
   DUP 4 = IF ." Очень крупные "       ELSE
              ." Ошибка "
   THEN THEN THEN THEN THEN DROP ;
```

Например:

```
1  МАРКИРОВКА Мелкие ok
```

Теперь мы можем определить слово РАЗМЕР-ЯИЦ, используя три наших собственных слова:

```
: РАЗМЕР-ЯИЦ  ( вес-на-дюжину -- )
   КАТЕГОРИЯ  DUP МАРКИРОВКА  УЧЕТ ;
```

Таким образом, выражение 23 РАЗМЕР-ЯИЦ выведет на вашем дисплее сообщение

```
Средние ok
```

и обновит счетчик яиц среднего размера.

Каким образом мы узнаем содержимое счетчиков в конце дня? Придется проверить по отдельности каждую ячейку массива, например, с помощью выражения 3 СЧЕТЧИК? (которое выведет число упакованных коробок с «крупными» яйцами). Однако можно

---

[1] *Для специалистов.* Более элегантный вариант этого определения приводится в следующей главе.

попытаться для печати результирующей таблицы за день в приведенном ниже формате определить свое собственное слово:

КОЛИЧЕСТВО	РАЗМЕР
1	Брак
112	Мелкие
132	Средние
143	Крупные
159	Очень крупные
0	Ошибка

Так как вы уже научились получать номера категорий, можно просто использовать цикл **DO** с номером категории в качестве индекса:

```
: СВОДКА   PAGE   ." Количество              Размер"   CR CR
   6 0 DO  I СЧЕТЧИК  @  5 U.R
   7 SPACES  I МАРКИРОВКА  CR       LOOP ;
```

(Выражение

```
I СЧЕТЧИК @  5 U.R
```

выбирает номер категории, подготовленный словом **I**, как индекс массива и выводит содержимое соответствующего элемента последнего в виде поля из пяти значений.)

ВЫЧЛЕНЕНИЕ ОПРЕДЕЛЕНИЙ

Рассмотрим теперь проблему разбиения применительно к определениям Форта. Мы только что привели пример, в котором разбиение определений упростило нам решение задачи.

Наше первое определение слова РАЗМЕР-ЯИЦ (см. гл. 4) сортировало яйца по весу и выводило на печать соответствующие категории яиц. В предыдущем примере мы разбили «сортировку» и «печать» на два отдельных слова. Вы можете использовать слово КАТЕГОРИЯ с целью выработки аргумента как для слова, инициирующего печать, так и для слова, осуществляющего подсчет (или для обоих вместе). Можно применить слово МАРКИРОВКА, обеспечивающее вывод на печать, и для слова РАЗМЕР-ЯИЦ, и для слова СВОДКА.

Ч. Мур, автор языка Форта, утверждает, что «хороший словарь Форта содержит большое число небольших определений. Недостаточно разбить некоторую задачу на небольшие фрагменты. Суть дела в том, чтобы выделить слова, которые можно *повторно* использовать». Например, в следующем рецепте

Взять банку с томатным соусом.

Открыть ее. Выложить томатный соус на сковороду.

Взять банку с грибами.

Открыть ее.

Выложить грибы на сковороду

вы можете «вычленить» действия: взять, открыть, выложить и объединить их в одном месте, так как они являются общими по отношению и к банке с томатным соусом, и к банке с грибами. После этого вы можете присвоить процессу в целом имя и в дальнейшем просто писать:

```
ТОМАТ ДОБАВИТЬ
ГРИБЫ ДОБАВИТЬ
```

и любой шеф-повар, окончивший «постфиксную» кулинарную школу, хорошо поймет, что вы имеете в виду.

Вычленение определений не только упрощает написание программы (и ее отладку), но и позволяет экономить память. Повторно используемое слово, например добавить, нужно определить только один раз. Чем сложнее программа, тем больше мы экономим при ее разбиении. Прежде чем покинуть птицеферму, приведем еще одно соображение по поводу стиля программирования на Форте. Вспомним наше определение слова РАЗМЕР-ЯИЦ:

```
: РАЗМЕР-ЯИЦ  ( вес-на-дюжину -- )
   КАТЕГОРИЯ  DUP МАРКИРОВКА  УЧЕТ ;
```

Слово КАТЕГОРИЯ доставляет значение, которое нам хотелось бы передать как слову МАРКИРОВКА, так и слову УЧЕТ, поэтому мы включаем сюда операцию **DUP**. Чтобы сделать определение более ясным, рискнем вынести из него **DUP** и поместим его в самое начало определения МАРКИРОВКА. Таким образом, можно написать:

```
: МАРКИРОВКА  ( номер-категории -- номер-категории)  и т.д.
: РАЗМЕР-ЯИЦ  ( вес-на-дюжину -- )
   КАТЕГОРИЯ  МАРКИРОВКА  УЧЕТ ;
```

где КАТЕГОРИЯ передает значение слову МАРКИРОВКА, а МАРКИРОВКА передает его слову УЧЕТ. Несомненно, этот вариант должен «сработать». Но впоследствии при определении СВОДКА мы вынуждены будем применить выражение I МАРКИРОВКА DROP вместо простого I МАРКИРОВКА.

Программирующим на Форте рекомендуется придерживаться следующего соглашения: там, где это возможно, слова должны уничтожать свои параметры. Вообще лучше помещать **DUP** в «вызывающем» определении (РАЗМЕР-ЯИЦ), чем в «вызываемом» (МАРКИРОВКА).

ОРГАНИЗАЦИЯ ЦИКЛА ПО МАССИВУ

Предлагаем вашему вниманию один из приемов работы с массивами. Лучше всего проиллюстрировать этот прием, написав на Форте наше собственное определение слова с именем **DUMP**. Оно применяется для вывода содержимого ряда адресов памяти. Форма использования этого слова следующая:

```
адрес длина DUMP
```

Например, мы можем ввести

```
СЧЕТЧИКИ 12 DUMP
```

и на печать будет выведено содержимое нашего массива СЧЕТЧИКИ, обеспечивающего нам подсчет яиц. Так как слово **DUMP** изначально было создано как средство программиста для вывода на печать содержимого участков памяти, то мы и получаем это содержимое либо байт за байтом, либо ячейку за ячейкой, в зависимости от вида адресации, применяемой в данном компьютере. В нашем варианте **DUMP** будет выводить ячейку за ячейкой.

Очевидно, что в рассматриваемом здесь примере слово **DUMP** содержит цикл **DO**. Вопрос заключается в том, что должно использоваться в качестве индекса. Хотя мы можем задействовать как индекс цикла непосредственно сам счетчик (0—6), все же для скорости лучше взять за индекс АДРЕС. Адрес массива СЧЕТЧИКИ будет начальным индексом для нашего цикла, а адрес плюс счетчик — верхней границей, например:

```
: DUMP  ( а длина -- )
   OVER + SWAP  DO  CR I @  5 U.R  2 +LOOP ;
```

Ключевым выражением здесь является фраза

```
OVER + SWAP
```

которая непосредственно предшествует **DO**.



Теперь начальный и конечный адреса находятся в стеке, готовые к использованию в качестве верхней границы и индекса цикла **DO**.

Выражение типа

```
OVER + SWAP
```

называется *клише*, поскольку оно выполняет некоторое единое действие с точки зрения более высокого уровня. Данное конкретное клише преобразует адрес и счетчик в форму аргументов оператора **DO**. Так как индексирование осуществляется по адресам, для того чтобы напечатать содержимое каждого элемента массива, поскольку мы находимся внутри цикла, достаточно просто ввести

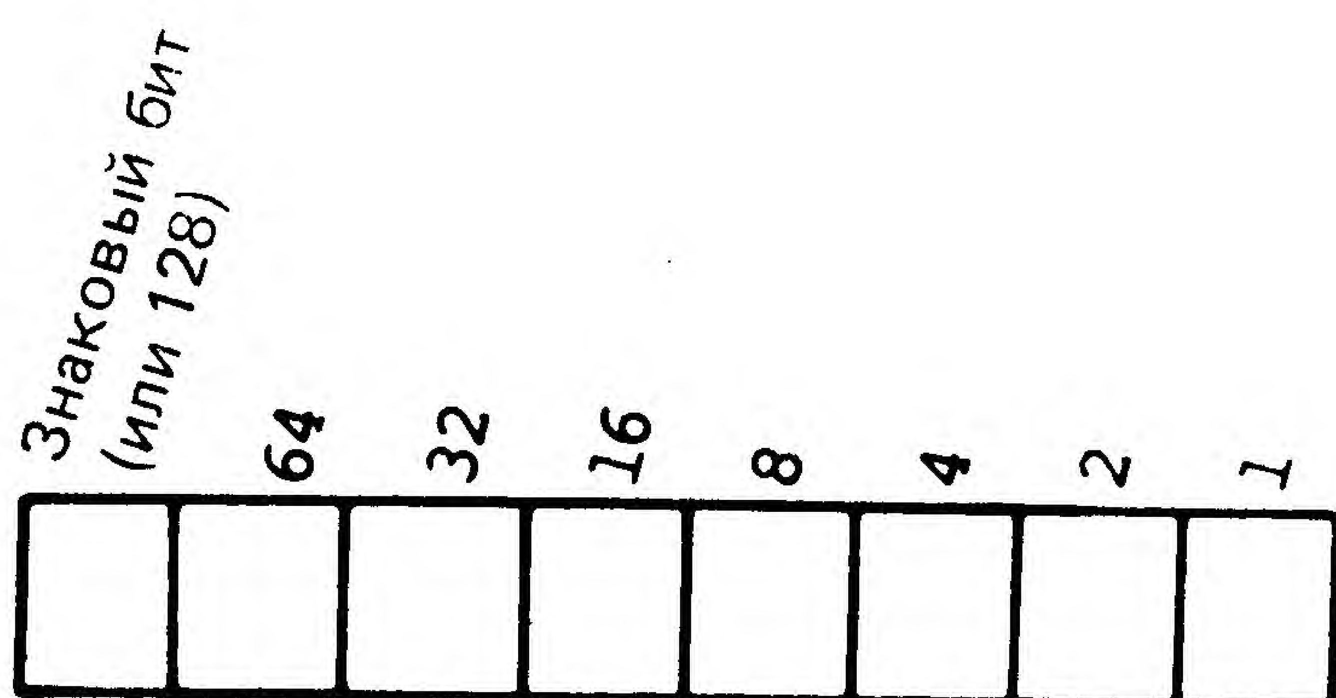
```
I @  5 U.R
```

Проверка байтов выполняется попарно (из-за того, что @ выбирает 16-разрядное значение), поэтому мы всякий раз увеличиваем индекс на *два* с помощью выражения

```
2 +LOOP
```

МАССИВЫ БАЙТОВ

Форт позволяет создать массив, каждый элемент которого содержит не полную ячейку, а один байт. Это полезно в тех случаях, когда вы запоминаете ряд чисел, представляемых восемью битами.



Диапазон значений 8-разрядного числа без знака — от 0 до 255. Байтовые массивы могут также служить для хранения строк символов в коде ASCII. Преимущество массива байтов перед массивом ячеек заключается в том, что при его применении вы можете иметь тот же объем данных при половинном объеме памяти.

Механизм использования байтового массива тот же, что и массива ячеек, за исключением двух положений:

1) вы не должны удваивать смещение, так как каждый элемент соответствует одному адресу;

2) слова ! и @ нужно заменить словами **C!** и **C @**. Этим словам, которые функционируют только с байтовыми значениями, дан префикс **C**, потому что обычно они обеспечивают доступ к символам в коде ASCII.

C!	(b a --)	Занесение 8-разрядного числа по заданному адресу.
C@	(a -- b)	Выборка 8-разрядного числа по заданному адресу.

ИНИЦИАЛИЗАЦИЯ МАССИВА

Во многих ситуациях требуется массив со значениями, которые во время выполнения прикладной программы никогда не меняются и могут быть загружены в этот массив во время его создания по аналогии с константами, создаваемыми с помощью **CONSTANT**. Для осуществления этих действий в Форте предусмотрено слово , (ЗАПЯТАЯ).

Предположим, вы хотите, чтобы значения массива ПРЕДЕЛЫ были постоянными. Тогда вместо выражения

```
VARIABLE ПРЕДЕЛЫ  8 ALLOT
```

необходимо ввести

```
CREATE ПРЕДЕЛЫ  220 ,  340 ,  170 ,  100 ,  190 ,
```

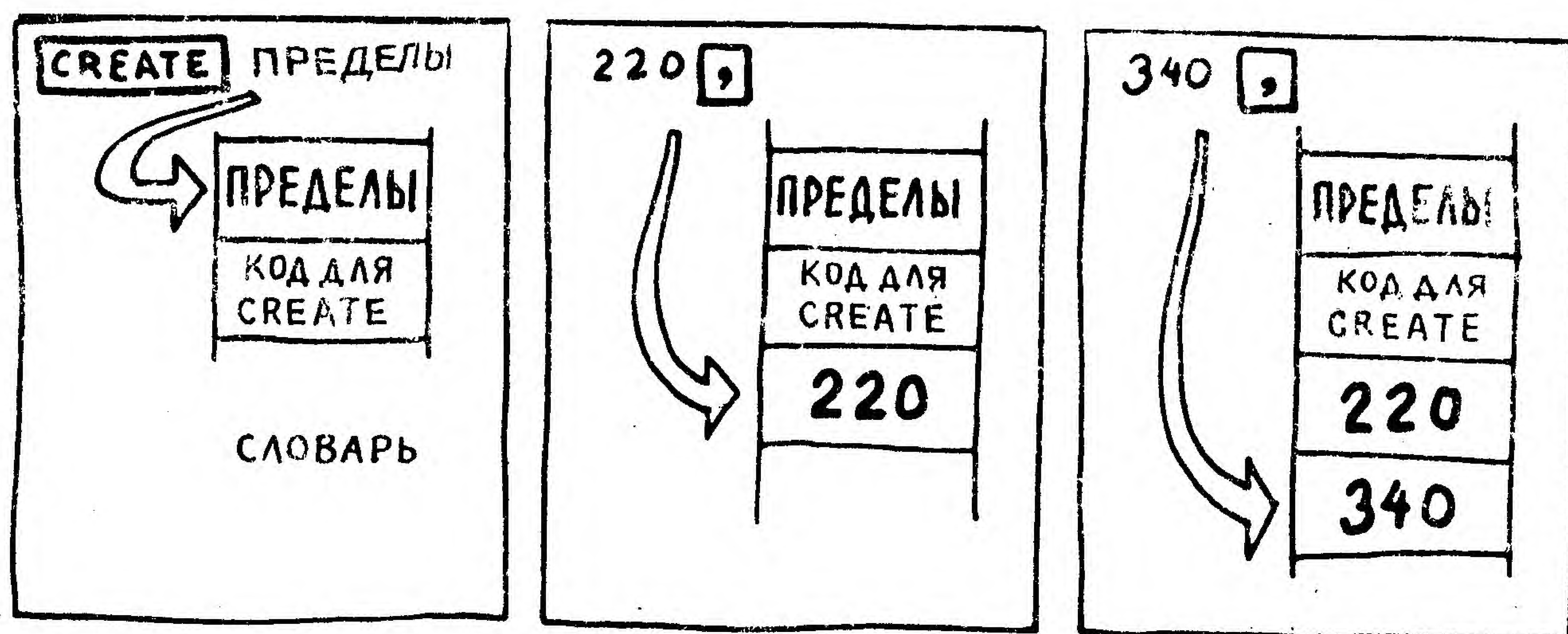
Как правило, приведенная выше строка должна вводиться из дискового блока, но ее можно вводить и непосредственно с экрана.

Напоминаем вам, что слово **CREATE** во время компиляции вносит новое имя в словарь, а во время выполнения доставляет адрес своего определения. Но оно не «выделяет» какое-то число байтов под значение.

Слово , берет число из стека и помещает его в массив. Поэтому всякий раз, когда вы записываете число и ставите за ним запятую, вы добавляете к массиву одну ячейку[1].

---

[1] *Для начинающих.* Укоренившиеся привычки грамотно писать фразы на естественном языке приводят к тому, что некоторые новички забывают написать последнюю запятую в строке. Помните, что «,» не разделяет числа, а компилирует их.



Доступ к элементам массива, созданного с помощью **CREATE**, обеспечивается точно так же, как и к элементам массива, созданного посредством **VARIABLE**, например:

```
ПРЕДЕЛЫ 2+ ? 340 ok
```

Вы даже можете помещать в этот массив новые значения так, как если бы работали с массивом, созданным с помощью **VARIABLE**, но только в том случае, кода вы не собираетесь пропустить свою прикладную программу через целевой компилятор.

Для того чтобы сформировать массив байтов, можно воспользоваться **C,**. Например, мы могли бы загрузить каждое из значений в нашем определении слова КАТЕГОРИЯ при сортировке яиц, следующим образом:

```
CREATE РАЗМЕРЫ  18 C,  21 C,  24 C,  27 C,  30 C,  255 C,
```

Это позволило бы переопределить слово КАТЕГОРИЯ посредством оператора цикла **DO**, а не рядом вложенных операторов **IF ... THEN**, в частности[1]:

---

[1] *Для тех, кто не любит докапываться до сути самостоятельно.* Суть дела состоит в следующем. Так как у нас пять возможных категорий, можно использовать номер категории как индекс для цикла. При выполнении каждого шага цикла мы сравниваем число из стека с элементом массива РАЗМЕРЫ, отстоящим от начала массива на смещение, равное текущему индексу цикла. Как только вес в стеке превысит значение очередного элемента массива, мы завершим цикл и с помощью **I** определим, сколько раз цикл выполнялся до того, как мы вышли из него. Поскольку это число представляет собой смещение в нашем массиве, то оно также является и номером категории.

```
: КАТЕГОРИЯ ( вес-на-дюжину -- номер-категории )
   6 0 DO  DUP  РАЗМЕРЫ I + C@
   < IF  DROP  I  LEAVE  THEN LOOP ;
```

Заметьте, что мы добавили к массиву максимальное значение (255), что дает возможность упростить наше определение для категории 5.

Этот новый вариант более изящен и более компактен по сравнению с прежним.

Ниже приводится перечень слов Форта, с которыми вы познакомились в данной главе.

VARIABLE xxx	(--) xxx: (-- a)	Создание переменной с именем xxx. Слово xxx при выполнении помещает в стек свой адрес.
!	(n a --)	Запоминание числа одинарной длины по заданному адресу.
@	(a -- n)	Замещение адреса его содержимым.
?	(a --)	Вывод значения по заданному адресу с последующим пробелом.
+!	(n a --)	Сложение числа одинарной длины с содержимым заданного адреса.
CREATE xxx	(--) xxx: (-- a)	Создание заголовка в словаре с именем xxx. Слово xxx при выполнении заносит в стек свой адрес.
ALLOT	(n --)	Резервирование в поле параметров слова, определенного последним, n дополнительных байтов.
C!	(b a --)	Занесение 8-разрядного числа по заданному адресу.
C@	(a -- b)	Выборка 8-разрядного числа по заданному адресу.
FILL	(a u b --)	Заполнение n байтов памяти, начиная с заданного адреса, значением b.
ERASE	(a u --)	Заполнение n байтов памяти, начиная с заданного адреса, нулями.
CONSTANT xxx	(n --) xxx:(-n)	Создание константы с именем xxx и значением n. Слово xxx при своем выполнении заносит в стек n.
FALSE	(-- f)	Занесение в стек логического значения ложь (0).
TRUE	(-- t)	Занесение в стек логического значения истина (-1).

Операции двойной длины:

2VARIABLE xxx	(--)	Создание переменной двойной длины с именем xxx. Слово xxx при выполнении помещает на стек свой адрес.
xxx:	(-- a)	
2CONSTANT xxx	(d --)	Создает константу двойной длины с именем xxx и значением d.
xxx:	(-- d)	Слово xxx при выполнении помещает в стек значение d.
2!	(d a --)	Запоминание числа двойной длины по заданному адресу.
2@	(a -- d)	Занесение в стек числа двойной длины, расположенного по заданному адресу.

Условные обозначения:

n, n1 ... - 16-разрядные числа со знаком;
a - адрес;
d, d1 ... - 32-разрядные числа со знаком;
u, u1 ... - 16-разрядные числа без знака;

ОСНОВНЫЕ ТЕРМИНЫ

Выборка. Выборка значения из заданного участка памяти.

Запоминание. Помещение некоторого значения в заданный участок памяти.

Индекс. Число, которое при обращении к массиву указывает относительную позицию его конкретного элемента. Умножая индекс на длину каждого элемента, мы получаем смещение, добавляемое к начальному адресу массива с тем, чтобы установить абсолютный адрес искомого элемента.

Инициализация. Присвоение переменной (или массиву) ее начальных значений прежде, чем остальная часть программы начнет выполняться.

Константа. Значение, которому присвоено имя. Это значение хранится в памяти и, как правило, не изменяется.

Массив. Ряд участков памяти с одним именем. Значения могут быть помещены в отдельные элементы массива и выбраны из них путем указания имени массива и добавления смещения к его адресу.

Переменная. Участок памяти с именем, в который (и из которого) можно последовательно заносить (и выбирать определенные значения.

Разбиение. Применительно к программированию на Форте — упрощение большой программы путем вычленения элементов, ко-

торые можно неоднократно использовать, и определения таких элементов как операций.

Смещение. Число, которое может быть добавлено к адресу начала некоторого массива для получения адреса требуемого участка памяти в пределах данного массива.

УПРАЖНЕНИЯ

8.1. а) Создайте два слова с именами ИСПЕКИ-ПИРОЖОК и СЪЕШЬ-ПИРОЖОК. Первое слово увеличивает число имеющихся пирожков на единицу, второе уменьшает это число на единицу и благодарит вас за пирожок. Но если пирожков в наличии нет, оно выводит: «Какой пирожок?». (Начните в предположении, что число пирожков равно нулю.)

```
СЪЕШЬ-ПИРОЖОК Какой пирожок?
ИСПЕКИ-ПИРОЖОК ok
СЪЕШЬ-ПИРОЖОК Спасибо! ok
```

б) Напишите слово с именем ЗАМОРОЗЬ-ПИРОЖКИ, которое берет все имеющиеся в наличии пирожки и добавляет их к числу пирожков, хранящихся в морозильной камере. Помните, что замороженные пирожки есть нельзя.

```
ИСПЕКИ-ПИРОЖОК ИСПЕКИ-ПИРОЖОК ЗАМОРОЗЬ-ПИРОЖКИ ok
ПИРОЖКИ ? 0 ok
ЗАМОРОЖЕННЫЕ-ПИРОЖКИ ? 2 ok
```

8.2. Определите слово с именем .BASE, которое выводит текущее значение переменной **BASE** в десятичной системе счисления. Проверьте это слово при первом же изменении переменной **BASE** на любом значении, отличном от 10. (Это не так просто, как может показаться.)

```
DECIMAL .BASE 10 ok
HEX .BASE 16 ok
```

8.3. Определите слово для форматизации чисел с именем М., которое выводит на печать число двойной длины с десятичной точкой. Положение десятичной точки внутри числа подвижно и зависит от значения некоторой переменной.

Если в вашей системе имеется для указания положения десятичной точки только что введенного числа переменная **DPL** или какая-либо другая, воспользуйтесь ею, например так:

```
2000.00 М. 2000.00 ok
```

В противном случае определите переменную с именем ДРОБНЫЕ и загружайте туда значения вручную:

```
2 ДРОБНЫЕ !
2000.00 М. 2000.00 ok
```

(В том случае, когда вводимое число не содержит десятичной точки, **DPL** вырабатывает —1 и в вершину стека помещается целое число одинарной длины. Для повышения надежности учтите при создании М. эту ситуацию.)

8.4. Для того чтобы иметь сведения о наличии цветных карандашей в вашем учреждении, создайте массив, каждая ячейка которого содержала бы счетчик карандашей одного конкретного цвета. Определите набор слов, такой, что выражение типа

```
КРАСНЫЕ КАРАНДАШИ
```

доставляет адрес ячейки, содержащей число красных карандашей и т. д., а затем присвойте этим переменным значения, чтобы набор карандашей был следующим:

```
23 КРАСНЫЕ КАРАНДАШИ
15 ГОЛУБЫЕ КАРАНДАШИ
12 ЗЕЛЕНЫЕ КАРАНДАШИ
 0 ЖЕЛТЫЕ КАРАНДАШИ
```

8.5. Создайте массив значений и напечатайте гистограмму (см. упражнения к гл. 5), где для каждого значения выводилась бы строка символов *. Для начала сформируйте массив из 10 элементов. Присвойте всем элементам массива значения в диапазоне от нуля до 70, после чего определите слово РИСУЙ, которое будет выводить строку, соответствующую каждому значению. На каждой строке должен выводиться номер элемента, а после него — число звездочек, равное содержимому данного элемента.

8.6. В данном упражнении мы используем переменные одинарной длины как массивы флагов, а константы — как битовые маски. Начните с определения констант, каждая из которых представляет отдельную битовую позицию (из четырех младших разрядов 16-разрядного значения):

```
ЖЕНЩИНА  СЕМЕЙНЫЙ  РАБОТАЕТ  ГОРОДСКОЙ
```

Теперь определите следующие константы, каждая из которых несет нули во всех битах:

```
МУЖЧИНА  ОДИНОКИЙ  НЕ-РАБОТАЕТ  НЕ-ГОРОДСКОЙ
```

Далее определите в виде переменных два имени, например:

```
VARIABLE ВАСЯ
VARIABLE МАША
```

Впоследствии в этих переменных будут содержаться атрибуты наших персонажей.

Затем определите слово с именем ОПИСАНИЯ, которое выбирает из стека значения четырех констант из восьми нами заведенных плюс адрес поля, отведенного для одного из наших персонажей:

```
ЖЕНЩИНА ОДИНОКИЙ РАБОТАЕТ ГОРОДСКОЙ  МАША ОПИСАНИЯ
```

Слово ОПИСАНИЯ заносит в область, отведенную персонажу, битовый шаблон, соответствующий атрибутам персонажа.

Наконец, определите слово СВЕДЕНИЯ, которое выбирало бы из стека имя персонажа и по битовому шаблону последнего выводило бы сведения о нем, например:

```
МАША СВЕДЕНИЯ женщина одинокий работает городской ok
```

8.7. Создайте программу, которая высвечивала бы поле для игры в «крестики-нолики» и давала возможность двум игрокам задавать свои ходы с помощью клавиатуры. Так, выражение 4 X! помещает X в клетку 4 (счет клеток ведется с единицы) и выводит на экран картинку:

```
   :   :
----------
X  :   :
----------
   :   :
```

После этого выражение 3 0! помещает 0 в клетку 3 и выводит следующую картинку:

```
   :   : 0
----------
X  :   :
----------
   :   :
```

Для того чтобы помнить содержимое игрового поля, примените массив байтов, где значение 1 соответствовало бы X, значение −1 — 0, а 0 — пустой клетке.

Так как некоторые компьютеры не позволяют программно подводить курсор к определенному месту экрана, мы будем просто заново генерировать поле для игры с помощью **CR**, **SPACE** и т. д.

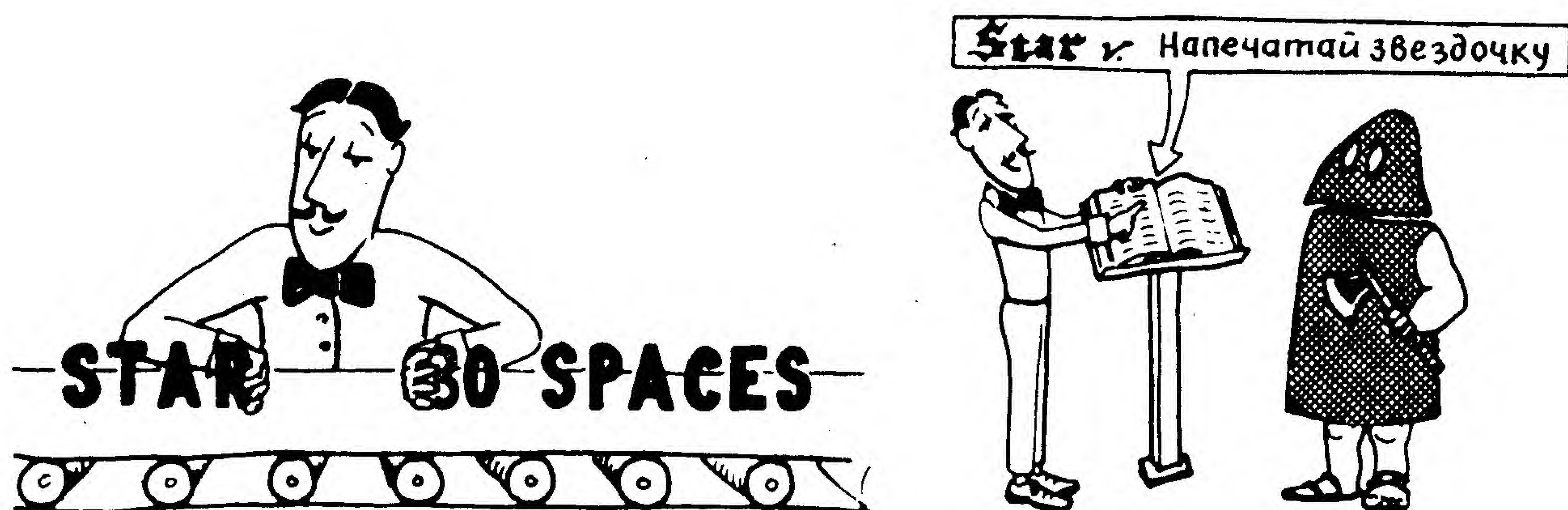
(*Замечание*: до тех пор, пока мы не приведем дополнительные сведения о словарях, избегайте обозначения чего-либо посредством «X», так как ваше имя может вступить в конфликт с редакторским X.)

Глава 9
ФУНКЦИОНИРОВАНИЕ ФОРТ-СИСТЕМЫ

В этой главе мы приоткроем «щелочку» и заглянем внутрь Форт-системы. Некоторое представление о том, что там (внутри) происходит, вы уже получили ранее, но, рискуя повториться, мы приведем здесь описание Форт-машины в целом, чтобы вы поняли, как все ее механизмы взаимодействуют друг с другом.

ПОИСК ПО СЛОВАРЮ

В первой главе было показано, как текстовый интерпретатор **INTERPRET** выбирает слова из входного потока и ищет их определения в нашем словаре. Если он находит слово, то исполняет его.



Посмотрим, из каких компонентов состоит текстовый интерпретатор, начнем с изучения слов поиска по словарю. Слово ' (апостроф) находит определение в словаре и помещает в стек *адрес* этого определения. Воспользуемся словом ВСТРЕЧА, которое мы определили в гл. 1, и запишем:

```
' ВСТРЕЧА U. 25520 ok
```

В результате получим адрес слова ВСТРЕЧА (вот и все, что должно произойти).

(На самом деле и **INTERPRET**, и апостроф используют для поиска по словарю примитив с именем **FIND** (ПОИСК).)

Слово ' имеет несколько применений. Так, с помощью выражения ': ВСТРЕЧА. вы можете узнать, было ли слово ВСТРЕЧА определено, фактически не выполняя его (при выполнении этого выражения будет либо напечатан адрес, либо выдан ответ: «?»).

Вам также может понадобиться адрес для того, чтобы вывести с помощью **DUMP** содержимое определения, например:

```
' ВСТРЕЧА  12 DUMP
```

Можно сочетать апостроф со словом **EXECUTE**. Вспомните, что текстовый интерпретатор, найдя слово, передает его адрес слову **EXECUTE.** То же самое можете делать и вы. Слово **EXECUTE** выполняет определение, адрес которого задается в стеке. Таким образом, вы можете ввести (следуя Стандарту-83):

```
' ВСТРЕЧА EXECUTE Привет я говорю на Форте ок
```

и получить тот же эффект, что и при выполнении одного слова ВСТРЕЧА, только более изощренным способом.

Слово **EXECUTE** не проверяет, является ли заданный адрес правильным. Вся ответственность ложится на вас. Неверный адрес почти всегда приводит к разрушению системы. Стандартом-83 определено, что апостроф оставляет в вершине стека правильный адрес для **EXECUTE**. К сожалению, интерфейс между апострофом и **EXECUTE** менялся от диалекта к диалекту на протяжении многих лет.

В приведенной ниже таблице излагаются правила вычисления адреса слова для **EXECUTE**. В первом столбце даются примеры использования апострофа в режиме интерпретации (что было продемонстрировано выше). Например, на фиг-Форте вы можете ввести:

```
' ВСТРЕЧА CFA EXECUTE
```

Пояснения к остальным столбцам — более краткие.

	1. Получение адреса для EXECUTE в режиме интерпретации	2. Получение адреса для EXECUTE из входного потока (внутри определения)	3. Получение адреса следующего слова внутри определения для передачи EXECUTE	Что помещает апостроф в стек
Фиг-Форт	' имя CFA	[COMPILE] ' CFA	' имя CFA	pfa
MMS-Форт	' имя 2—	[COMPILE] ' 2—	' имя 2—	pfa
Стандарт-79	' имя CFA или FIND имя	[COMPILE] ' CFA	' имя CFA	pfa
полиФорт	' имя	'	['] имя	pfa
Стандарт-83	' имя	'	['] имя	cfa

ВЕКТОРНЫЕ ВЫЧИСЛЕНИЯ

Векторное вычисление, хотя и воспринимается как нечто трудоемкое, на самом деле осуществляется довольно просто. Вместо того чтобы выполнять какое-либо определение *непосредственно* (см. выражение ' ВСТРЕЧА EXECUTE), мы выполняем его *косвенно*, запоминая адрес определения в некоторой переменной, а затем выполняя содержимое этой переменной:

```
VARIABLE УКАЗАТЕЛЬ       ( участок для хранения исполнительного вектора)
' ВСТРЕЧА УКАЗАТЕЛЬ !    ( указатель ссылается на слово ВСТРЕЧА )
УКАЗАТЕЛЬ @ EXECUTE      ( выполнение фрагмента,на который ссылается
                           УКАЗАТЕЛЬ).
```

(Для вычисления адреса, который засылается в УКАЗАТЕЛЬ, примените соответствующее выражение из столбца 1 приведенной выше таблицы). В ряде систем есть слово **@EXECUTE**, которое эквивалентно выражению " @ EXECUTE", но выполняется более эффективно.

Вы можете попытаться сделать следующий пример самостоятельно:

```
1 VARIABLE 'ФРАЗА  ( вектор)
2 : ФРАЗА   'ФРАЗА @ EXECUTE ;   ( векторизуемое определение)
3 : ПРИВЕТ   ." Привет " ;
4 : ДО-СВИДАНИЯ   ." До свидания " ;
5
6 ' ПРИВЕТ   'ФРАЗА !   ( инициализация вектора)
```

В строке 1 вы определяете переменную с именем 'ФРАЗА. Она будет вашим указателем. В строке 2 специфицируется слово

ФРАЗА для выполнения определения, адрес которого находится в указателе 'ФРАЗА. В строках 3 и 4 создаются слова, выводящие сообщения: «Привет» и «До свидания». В строке 6 вы загружаете адрес слова ПРИВЕТ в указатель 'ФРАЗА по правилам, изложенным в столбце 1 вышеприведенной таблицы.

Теперь если выполнить слово ФРАЗА, то получится следующее:

```
ФРАЗА Привет ок
```

Если вы альтернативно выполните выражение

```
' ДО-СВИДАНИЯ 'ФРАЗА !
```

тем самым запомнив адрес ДО-СВИДАНИЯ в переменной 'ФРАЗА, то результат будет таким:

```
ФРАЗА До свидания ок
```

Апостроф в имени ' ФРАЗА означает, что данное имя является указателем при векторном вычислении — таковы соглашения в Форте. Так как апостроф при выполнении выдает адрес, то префикс соответствует адресу слова ФРАЗА.

Мы можем заставить само слово ФРАЗА выполнять все, что захотим, даже слова (ПРИВЕТ и ДО-СВИДАНИЯ), которые определены после слова ФРАЗА. Следовательно, апостроф обеспечивает один из способов реализации *ссылок вперед*. Ссылки вперед возникают в тех случаях, когда при определении некоторого слова приходится обращаться к другому слову, еще не определенному. В Форте нет естественных средств для программирования такой ситуации, да и в большинстве ситуаций вы можете просто по-иному расположить загружаемые определения. Но иногда переупорядочивание определений невозможно и вам без ссылок вперед не обойтись. В нашем примере на строке 6 происходит завершение реализации ссылки вперед.

Более общее применение векторных вычислений — изменение выполняемых действий некоторого слова уже *после того, как оно скомпилировано*. Определения слов, осуществляющих интерфейс, таких, как слова управления видеодисплеем, принтером, дисководом, часто делают векторными. Векторизация позволяет хранить имена для перечисленных функций в предварительно скомпилированной части Форт-системы и в то же время организовать свои собственные варианты их исполнения. Поскольку Форт-система сама использует вектора, ваши изменения будут влиять и на нее. Вы можете заставить систему после вывода «ok» сделать возврат каретки на любом терминале или принтере, к которому есть доступ.

АПОСТРОФ В ОПРЕДЕЛЕНИИ

Согласно Стандарту-83 апостроф всегда пытается найти следующее слово во входном потоке. Что произойдет, если мы поместим апостроф внутрь какого-либо определения? При исполнении такого определения апостроф будет искать следующее слово из входного потока. Таким образом, мы можем определить:

```
: СКАЖИ   ( имя  ( -- )   ' 'ФРАЗА ! ;
```

(Если у вас иная система, обратитесь к столбцу 2 вышеприведенной таблицы.) Необычный стековый комментарий означает, что слово СКАЖИ будет «заглядывать» вперед по входному потоку в поисках очередного слова.

Теперь можно ввести:

```
СКАЖИ ПРИВЕТ ok
ФРАЗА Привет ok
```

или

```
СКАЖИ ДО-СВИДАНИЯ ok
ФРАЗА До свидания ok
```

Апостроф в слове СКАЖИ осуществляет поиск имени определенных слов ПРИВЕТ и ДО-СВИДАНИЯ во входном потоке во время *выполнения* слова СКАЖИ. Во время *определения* этого слова апостроф ничего не делает (разве что позволяет себя компилировать).

А как быть, если нужно специфицировать посредством апострофа адрес следующего слова в определении? Для этого имеется слово ['], которое применяется вместо слова ', например[1]:

```
: ПРИХОДЯ   ['] ПРИВЕТ  'ФРАЗА ! ;
: УХОДЯ   [']  ДО-СВИДАНИЯ 'ФРАЗА ! ;
```

Введите следующий текст:

```
ПРИХОДЯ ok
ФРАЗА Привет ok
УХОДЯ ok
ФРАЗА До свидания ok
```

---

[1] *Для пользователей небольших систем.* Если на вашей клавиатуре нет клавиши «[» или «]», то в документации по Форт-системе должна быть указана замена.

В столбце 3 приведенной выше таблицы изложены правила выполнения этих действий на каждом из диалектов Форта. Далее дается список команд, которые мы уже рассмотрели.

' xxx	(-- a)	Осуществляется поиск в словаре адреса слова xxx (следующего слова во входном потоке).
[']	период-компиляции: (--) период-выполнения: (-- a)	Используется только в определении через двоеточие. Компиляция адреса следующего слова в определении как литерала.
EXECUTE	(a --)	Исполнение элемента словаря, адрес поля параметров которого находится на стеке.
@EXECUTE	(a --)	Исполнение элемента словаря, адрес которого является содержимым a. Если по адресу a находится нуль, @EXECUTE ничего не выполняет.

СТРУКТУРА СЛОВАРНОЙ СТАТЬИ

Все определения, независимо от того, были ли они специфицированы посредством :, **VARIABLE**, **CREATE** или любого другого *определяющего слова*, состоят из следующих полей[1]:

- поля имени;
- поля связи;
- поля кода;
- поля параметров.

Точное представление перечисленных полей в памяти зависит от реализации. И хотя Стандарт-83 предписывает полю имени «естественную длину» (в словаре запоминаются все символы имени — максимально 31), в наших примерах будет использоваться вариант с тремя хранимыми символами имени, так как это легче объяснить.

Ниже показано на примере переменной ДАТА, как рассматриваемые поля располагаются в памяти. На приведенной ниже схеме каждая строка представляет ячейку словаря.

---

[1] *Для систем, функционирующих на Форт-процессорах.* Приводимая структура элементов словаря и техника вызова процедур относится к виртуальным Форт-системам, реализованным на обычных процессорах с применением косвенного шитого кода, а не к системам, где Форт-процессор реализован непосредственно в кремнии. (Но, изучив материал настоящей главы, вы скорее поймете, как работает Форт-машина.)

Бит использования во время компиляции

Предыдущее (определение)

Имя: 4 Д / А Т

Связь

Указатель на код

Поле параметров

Поле имени. В нашем примере первый байт этого поля содержит число символов полного имени определяемого слова (в слове ДАТА четыре буквы). В следующих трех байтах хранятся представления первых трех букв имени определяемого слова в коде ASCII. В системах, где сохраняются только три символа имени, — это вся информация, которая используется апострофом или ['] для сравнения имени определения со словом из входного потока.

(Заметьте, что знаковый бит байта, в котором расположен счетчик символов, используется во время компиляции и показывает, будет ли данное слово исполняться во время компиляции или просто компилироваться в новое определение. Более подробно об этом идет речь в гл. 11.)

Поле связи. Ячейка *связи* содержит адрес предыдущего определения в словарном списке и применяется при поиске по словарю. В несколько упрощенном виде можно представить организацию связи следующим образом. Всякий раз, добавляя в словарь новое слово, компилятор заносит в поле связи указатель адреса предыдущего определения. На рисунке он заносит в поле связи слова КУЛИНАРИЯ указатель определения АВТОМОБИЛЬ.



Свой поиск апостроф или ['] начинает с самого последнего занесенного в словарь слова и просматривает «цепочку» в обратном направлении с помощью адресов, находящихся в ячейках связи.

Поле связи первого определения в словаре содержит нуль, который предписывает апострофу прекратить поиск — искомого слова в словаре нет.



Поле кода. Следующим полем является поле кода. Адрес, содержащийся в этом поле, позволяет отличать переменную от константы или от определения через двоеточие. Выполнение Форт-системой некоторого слова заключается в исполнении процедуры на машинном языке, на которую есть ссылка в данном поле. Например, в случае переменной указатель ссылается на код, который помещает адрес этой переменной в стек, в случае константы — на код, который помещает в стек ее содержимое, а в случае определения через двоеточие — на код, выполняющий оставшуюся часть этого определения.

Код, на который происходит ссылка, называется *кодом периода выполнения*, потому что он используется при выполнении слова данного вида (а не тогда, когда оно определяется или компилируется).

Все переменные имеют свой один и тот же указатель кода, а все константы — свой. Все определения через двоеточие — то же и т. д.

Поле параметров. За полем указателя кода следует поле параметров. В случае переменных и констант поле параметров представляет собой только одну ячейку, в случае **2CONSTANT** или **2VARIABLE** — две, в случае массива — столько, сколько вы пожелаете, а в случае определения через двоеточие длина поля параметров зависит от длины данного определения, о чем и пойдет речь в следующем разделе.

VARIABLE ДАТА

4	Д
А	Т
Поле связи	
Указатель на код	
12	

VARIABLE

Код периода выполнения (при выполнении помещает адрес переменной в стек)

220 CONSTANT ПРЕДЕЛ

6	П
Р	Е
Поле связи	
Указатель на код	
220	

CONSTANT

Код периода выполнения (при выполнении помещает в стек содержимое константы)

: РАЗМЕР – ЯИЦ

10	Р
А	З
Поле связи	
Указатель на код	
(Собственно определение)	

:

Код периода выполнения (при выполнении исполняет слова, входящие в состав данного определения)

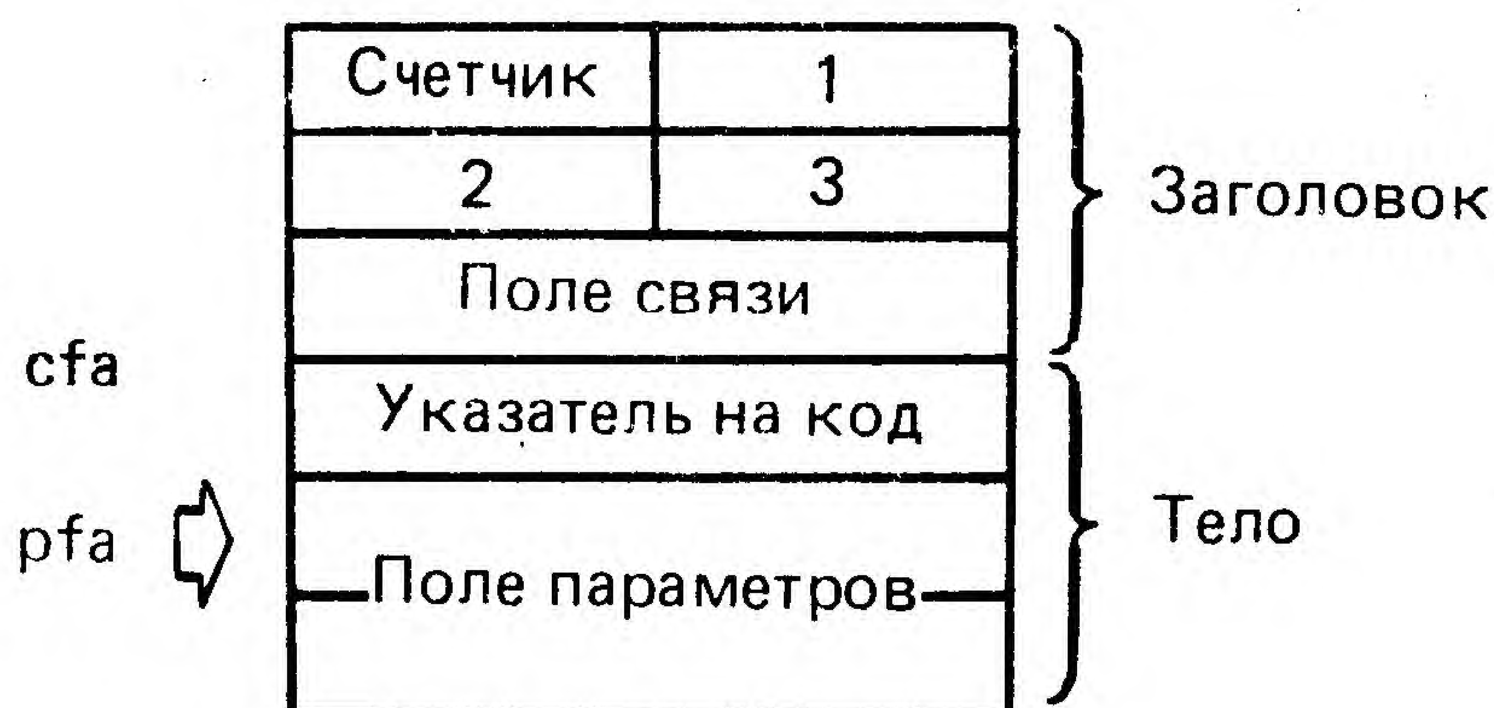
VARIABLE 'ФРАЗА

6	'
Ф	Р
Поле связи	
Указатель на код	
2AE4	

АДРЕСАЦИЯ ПОЛЕЙ

При изучении полей, составляющих структуру словаря, важно понять различие между адресами этих полей и их *содержимым*. По соглашению адрес, по которому содержится указатель кода, называется *адресом поля кода* (cfa). Следовательно, cfa слов содержат указатель кода периода их выполнения.

Адрес первой ячейки, в которой хранится параметр, называется *адресом поля параметров* (pfa).



Адрес, помещаемый в стек апострофом, в одних реализациях является cfa, в других — pfa. Стандарт предусматривает слово **>BODY**, осуществляющее переход от cfa к pfa.

Таким образом, имеется возможность (хотя это далеко не «стандарт» и не может быть рекомендовано к применению) изменять значение существующей константы, например:

```
n ' ПРЕДЕЛ >BODY !
```

Если ваша система отличается от той, с которой мы работаем, обращайтесь к столбцу 4 приведенной ранее таблицы, где излагаются правила вычисления адреса для **EXECUTE**.

Между прочим поля имени и связи часто называют заголовком элемента, а поля указателя кода и параметров — его телом.

СТРУКТУРА ОПРЕДЕЛЕНИЯ ЧЕРЕЗ ДВОЕТОЧИЕ

Если формат заголовка и поля указателя кода одинаков для всех типов определений в конкретной системе, то формат поля параметров меняется от типа к типу. Рассмотрим поле параметров определения через двоеточие.

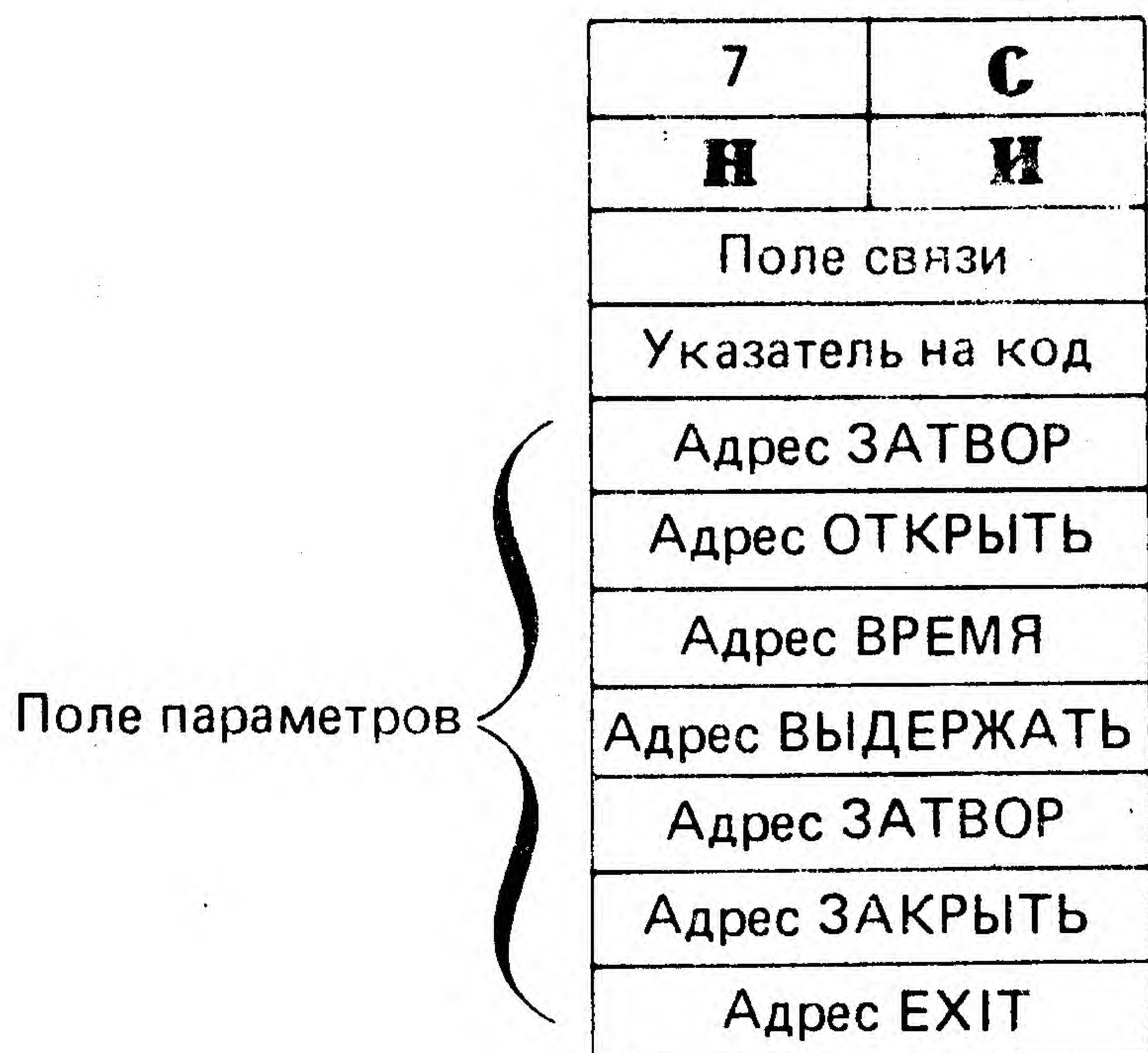
Поле параметров определения через двоеточие содержит *адреса* уже определенных слов, составляющих данное определение[1]. Ниже приводится элемент словаря для определения слова СНИМОК, которое выглядит следующим образом:

```
: СНИМОК   ЗАТВОР ОТКРЫТЬ  ВРЕМЯ ВЫДЕРЖАТЬ  ЗАТВОР ЗАКРЫТЬ ;
```

При выполнении слова СНИМОК определения, расположенные в последующих адресах, выполняются в порядке очереди. Механизм, который читает список адресов и выполняет определения, расположенные по каждому адресу, называется *адресным интерпретатором*. Слово ; в конце определения компилирует адрес слова с именем **EXIT**. Как видно из рисунка, адрес **EXIT** расположен в последней ячейке элемента словаря. Адресный интерпретатор выполнит слово **EXIT** тогда, когда он подойдет к его адресу, точно так же, как он выполняет остальные слова определения. **EXIT** завершает выполнение адресного интерпретатора, что будет показано в следующем разделе.

---

[1] *Для специалистов.* Адреса, составляющие тело определения через двоеточие, как правило, указывают поле кода, а не поле параметров (т. е. cfa, а не pfa).



ВЛОЖЕННЫЕ УРОВНИ ВЫЧИСЛЕНИЙ

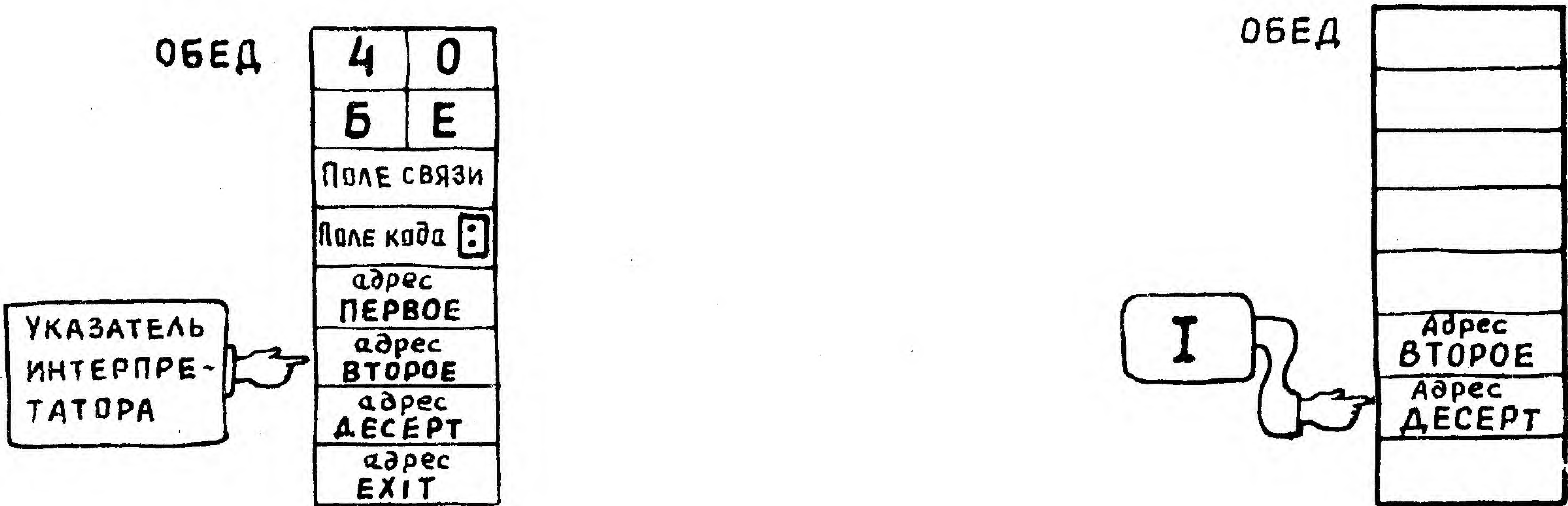
Функция слова **EXIT** заключается в том, чтобы возобновить процесс вычислений в определении более высокого уровня (по порядку вложенности), из которого была сделана ссылка на текущее определение. Рассмотрим упрощенную схему работы этого механизма. Предположим, что ОБЕД состоит из трех блюд:

```
: ОБЕД   ПЕРВОЕ ВТОРОЕ ДЕСЕРТ ;
```

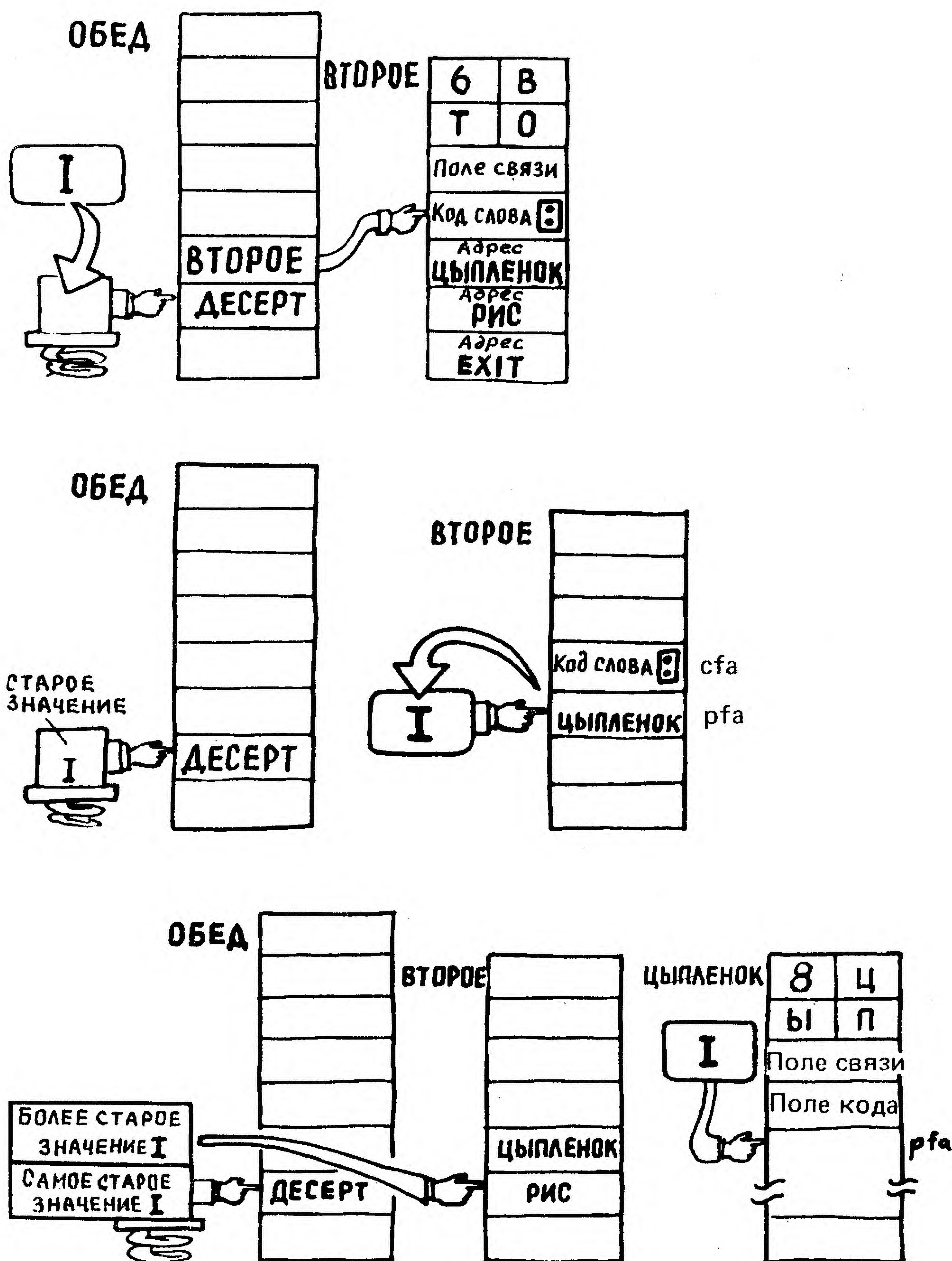
причем сегодня на ВТОРОЕ подается только

```
: ВТОРОЕ   ЦЫПЛЕНОК РИС ;
```

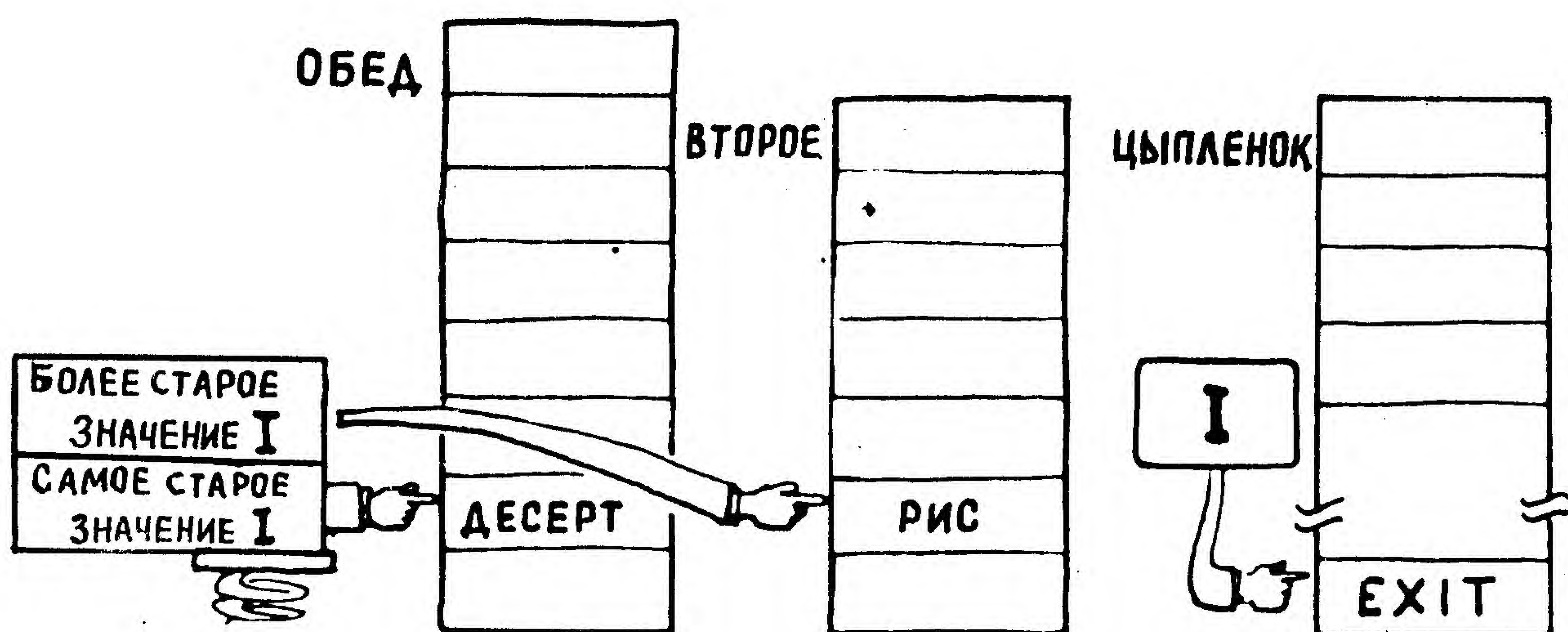
Мы находимся на стадии выполнения слова ОБЕД и только что покончили с блюдом ПЕРВОЕ. Указатель, которым пользуется адресный интерпретатор, называется *указателем интерпретатора* (**I**). Так как за блюдом ПЕРВОЕ следует ВТОРОЕ, наш указатель интерпретатора указывает ячейку, содержащую адрес слова ВТОРОЕ. Прежде чем перейти к выполнению слова ВТОРОЕ, увеличим значение указателя интерпретатора настолько, чтобы при возврате он указывал бы на ДЕСЕРТ.



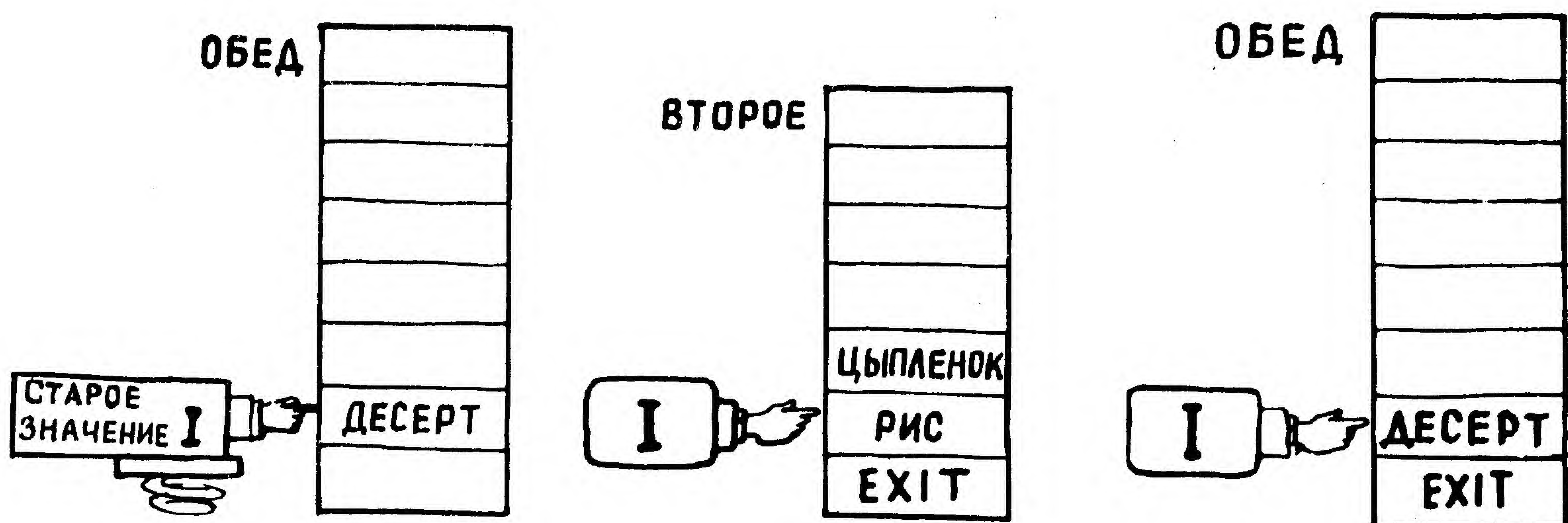
Теперь приступим к выполнению слова ВТОРОЕ и начнем с выполнения кода слова ВТОРОЕ, т. е. кода, на который указывает поле кода и который является общим для всех определений через двоеточие. Этот код выполняет две функции: вносит содержимое указателя интерпретатора в стек возвратов, а затем помещает адрес своего поля параметров (pfa) в указатель интерпретатора. Теперь указатель интерпретатора отсылает нас к слову ЦЫПЛЕНОК.



Итак, интерпретатор адреса готов приняться за цыпленка. Однако прежде, как и в случае со словом ВТОРОЕ, увеличим указатель настолько, чтобы он при возврате показывал на РИС. После этого код слова ЦЫПЛЕНОК заносит указатель в стек возвратов и помещает pfa слова ЦЫПЛЕНОК в указатель интерпретатора.



По мере того как мы наслаждаемся нашим аппетитным цыпленком, последовательно исполняется определение, его составляющие. Рано или поздно, обрабатывая слово ЦЫПЛЕНОК, мы подойдем к **EXIT.** Слово **EXIT** берет число из вершины стека возвратов и вносит его в указатель интерпретатора. Далее интерпретатор адреса продолжает процесс, выполняя слово РИС, покончив с последним. Несомненно, в конце концов, **EXIT** в слове ВТОРОЕ поместит значение из стека возвратов в указатель интерпретатора, и мы с вами созреем для десерта.



ЕЩЕ ОДИН ВАРИАНТ ИСПОЛЬЗОВАНИЯ СТЕКА ВОЗВРАТОВ

Итак, вы уже знаете, каким образом Форт-система хранит адреса возврата в стеке возвратов, и вам понятно, почему при хранении в этом стеке временных переменных нужна особая осторожность. При выполнении следующего определения

```
: ТЕСТ   3 >R  CR CR CR ;
```

в стек возвратов помещается значение 3, затем три раза осуществляется возврат каретки и происходит возврат, но куда? По адресу 3 нет никакого слова, поэтому скорее всего вероятно разрушение системы.

В большинстве Форт-систем во время выполнения цикла **DO** информация об индексе и границе хранится в стеке возвратов (и

не всегда в том виде, в котором вы ожидаете). Именно поэтому в пределах цикла **DO LOOP** использование операций **>R** и **R>** должно быть симметричным. Другими словами, вы имеете право писать так:

```
... >R ... DO ... LOOP ... R> ... ;
```

и не должны писать иначе, например:

```
... >R ... DO ... R> ... LOOP ... ;
```

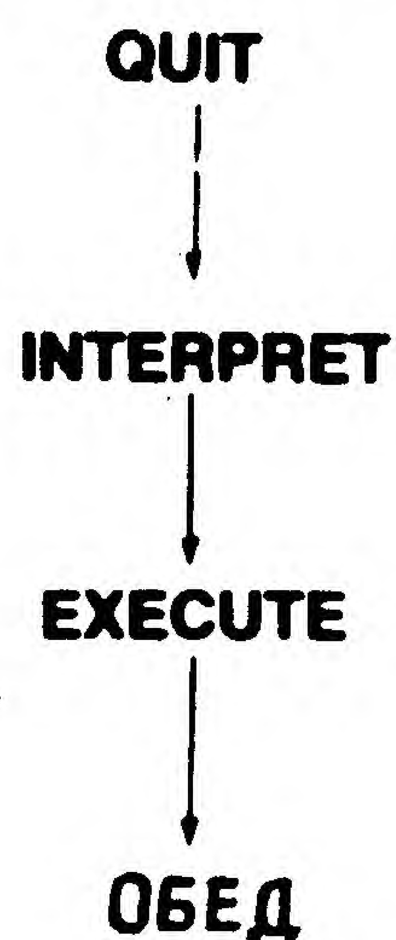
Кроме того, если вы внутри тела цикла поместите в стек возвратов временное значение, то слово выборки индекса цикла **I** не будет выполняться правильно:

```
... DO ... >R ... I ... R> ... LOOP ;
```

ВЫХОД НА ВЕРХНИЙ УРОВЕНЬ

Вам, вероятно, интересно узнать, что произойдет после того, как мы исполнили последний **EXIT** в слове ОБЕД. Чей адрес возврата находится в стеке? Куда предстоит вернуться?

Напомним, что слово ОБЕД было запущено на выполнение словом **EXECUTE**, которое является компонентой слова **INTERPRET**. Последнее осуществляет циклический процесс непрерывной проверки входного потока. Допустим, вы уже отобедали, и во входном потоке ничего нет для интерпретирования.



В этом случае при выполнении **EXIT** слова **INTERPRET** вы выходите на определение самого верхнего уровня с именем **QUIT**. В упрощенной форме **QUIT** выглядит следующим образом:

```
: QUIT   BEGIN  RESET QUERY  INTERPRET  ." ok" CR
   FALSE UNTIL ;
```

где RESET очищает стек возвратов, а QUERY настраивает входной поток на буфер входного текста.

(Определение **QUIT** в вашей системе может отличаться от приведенного.) Как видите, после слова **INTERPRET** следует сообщение точки — кавычки «ok» и **CR**, что и должно появиться на экране после завершения интерпретации. Далее идет выражение FALSE UNTIL, которое, безусловно, обеспечивает возврат в начало цикла, где вы очищаете стек возврата и снова ожидаете ввода.

Если исполнить **QUIT** на любом уровне выполнения, то немедленно прекратится вычисление по нашей программе и принудительно начнется выполнение цикла **QUIT**. Стек возвратов очистится (независимо от того, сколько в нем находится уровней адресов возврата, после чего вы никогда больше не сможете воспользоваться ни одним из них), а система будет ожидать ввода. Теперь вам понятно, почему слово **QUIT** может применяться для того, чтобы сообщение «ok» не выдавалось.

Другие способы использования QUIT. Еще два важных слова обращаются к **QUIT: ABORT**, которое, кроме всего прочего, очищает стек данных, и **ABORT″**. Второе слово

- выбирает из стека флаг и определяет по нему, нужно ли выполнять **ABORT** (при истинном значении нужно);
- перед аварийным завершением выдает сообщение, предписанное вами для конкретной аварийной ситуации.

Слово **ABORT″** было введено в конце гл. 4. Как правило, оно применяется в начале определений пользовательских слов (наивысшего уровня) для того, чтобы проверить хотя бы наличие аргументов в стеке, требуемых для выполнения этих слов. Например, вы можете написать следующее определение:

```
: LIST  ( n -- )  ГЛУБИНА-СТЕКА 1 < ABORT" Нет номера блока " LIST ;
```

или в более общем виде:

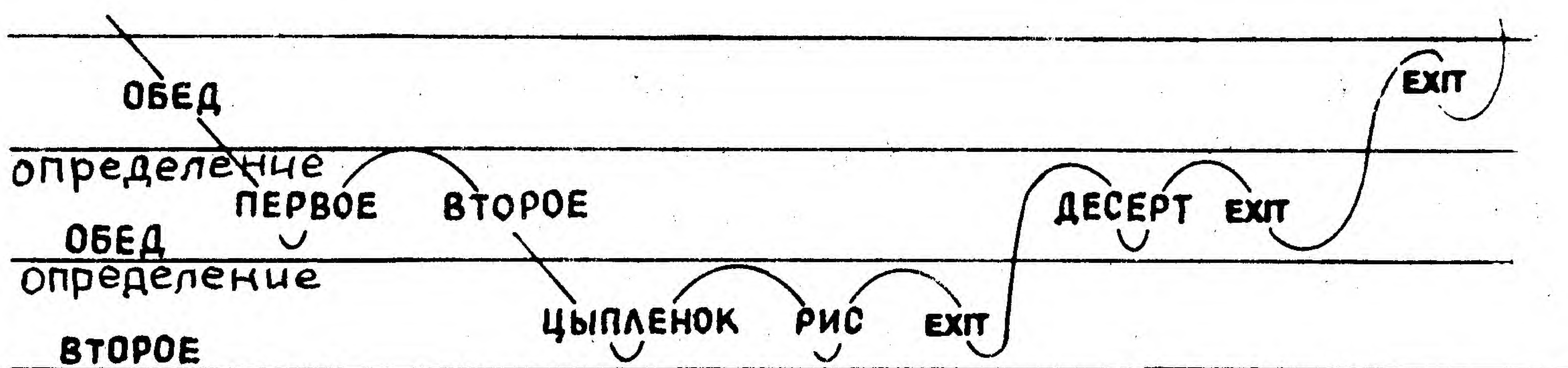
```
: АРГУМЕНТЫ  ( n -- )  ГЛУБИНА-СТЕКА < NOT ABORT" Значения? ;
```

что может быть использовано так:

```
: LIST  ( n -- )  1 АРГУМЕНТЫ  LIST ;
```

ПРОИЗВОЛЬНОЕ ИЗМЕНЕНИЕ ПОСЛЕДОВАТЕЛЬНОСТИ ВЫПОЛНЕНИЯ СЛОВ

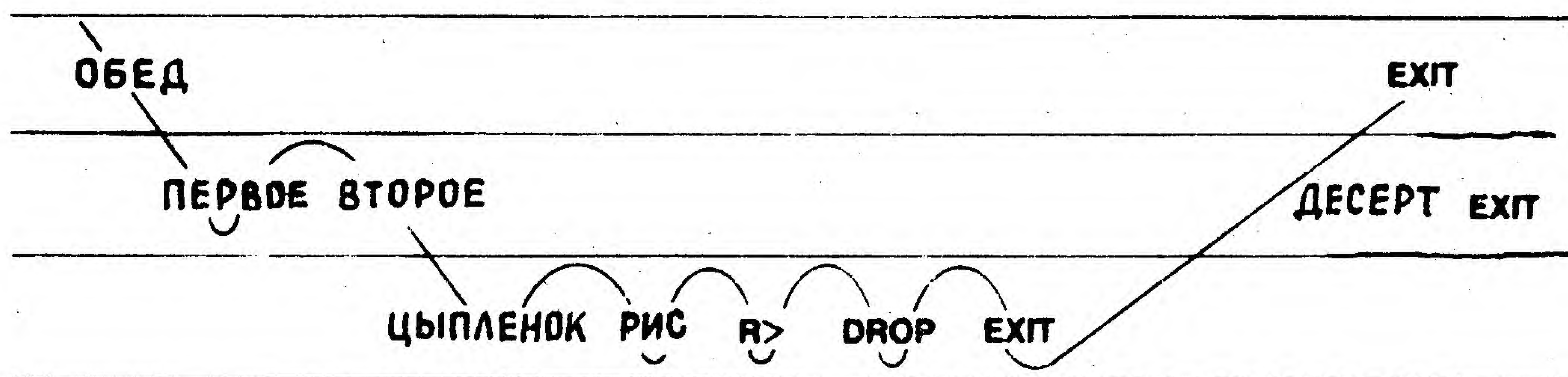
Существует возможность опустить один уровень исполнения, просто удалив один адрес из стека возвратов. В качестве примера рассмотрим три уровня исполнения, связанных со словом ОБЕД:



Предположим, что мы изменили определение ВТОРОЕ:

```
: ВТОРОЕ   ЦЫПЛЕНОК РИС  R> DROP ;
```

Выражение „R> DROP" удалит из стека возвратов адрес возврата в слове ДЕСЕРТ, который был туда помещен перед выполнением слова ВТОРОЕ. Если перезагрузить эти определения и выполнить слово ОБЕД, то **EXIT** третьего уровня обеспечит возврат непосредственно на первый уровень. Мы «съедим» ПЕРВОЕ, ЦЫПЛЕНКА и РИС, но останемся без ДЕСЕРТА:

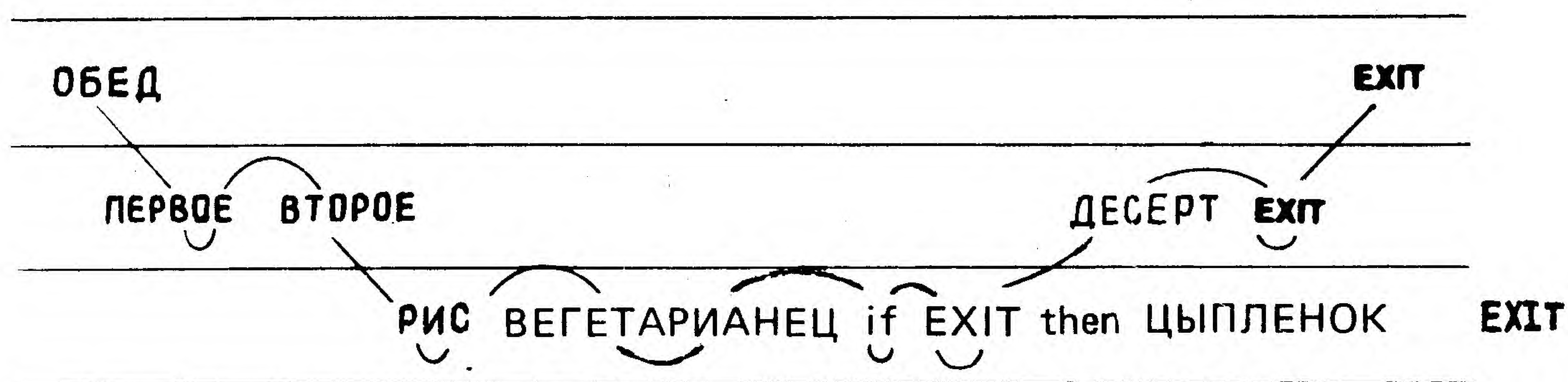


Применять в прикладной программе выражения "R > DROP" нежелательно, поскольку это противоречит принципам структурного программирования. Тем не менее данное выражение иногда позволяет упростить решение задачи. Мы не будем здесь приводить аргументы «за» и «против», однако изложенное должно послужить вам предостережением.

Как недавно упоминалось, слово **EXIT** удаляет адрес возврата из вершины стека возвратов и помещает его в указатель интерпретатора. Интерпретатор адреса, который ориентируется по содержимому указателя интерпретатора, начинает поиск на следующем верхнем уровне. Можно включить EXIT в середину определения. Например, если бы мы переопределили слово ВТОРОЕ:

```
: ВТОРОЕ   РИС  ВЕГЕТАРИАНЕЦ IF EXIT THEN  ЦЫПЛЕНОК ;
```

и при этом были бы вегетарианцами, то нам пришлось бы выполнить **EXIT** после слова РИС, пропустить слово ЦЫПЛЕНОК и перейти сразу к ДЕСЕРТУ.



Приведенное выше определение эквивалентно следующему:

```
: ВТОРОЕ   РИС  ВЕГЕТАРИАНЕЦ NOT  IF ЦЫПЛЕНОК THEN ;
```

Вы не имеете права использовать внутри оператора цикла **DO** слово **EXIT**, так как это слово удаляет из стека один из аргументов, занесенных в него оператором **DO** (вместо того, чтобы удалить из стека возвратов адрес возврата)!

Вы только что видели результат удаления адреса возврата из стека возвратов. Приведем еще один пример — занесение в стек возвратов лишнего адреса (вам, может быть, придется привести адреса cfa и pfa в соответствии с вашей системой):

```
: ПРИВЕТ   ." Привет " ;
: ДО-СВИДАНИЯ   ." До свидания " ;
' ДО-СВИДАНИЯ >BODY >R   ПРИВЕТ
```

Сначала с помощью апострофа выбираем адрес слова ДО-СВИДАНИЯ и помещаем его в стек возвратов — пусть Форт-система «думает», что это адрес возврата. Затем инициируем слово ПРИВЕТ, которое выдает свое приветствие. В конце концов, Форт-система обращается к стеку возвратов за следующим адресом и, выбрав его, выполняет ДО-СВИДАНИЯ — после слова привет:

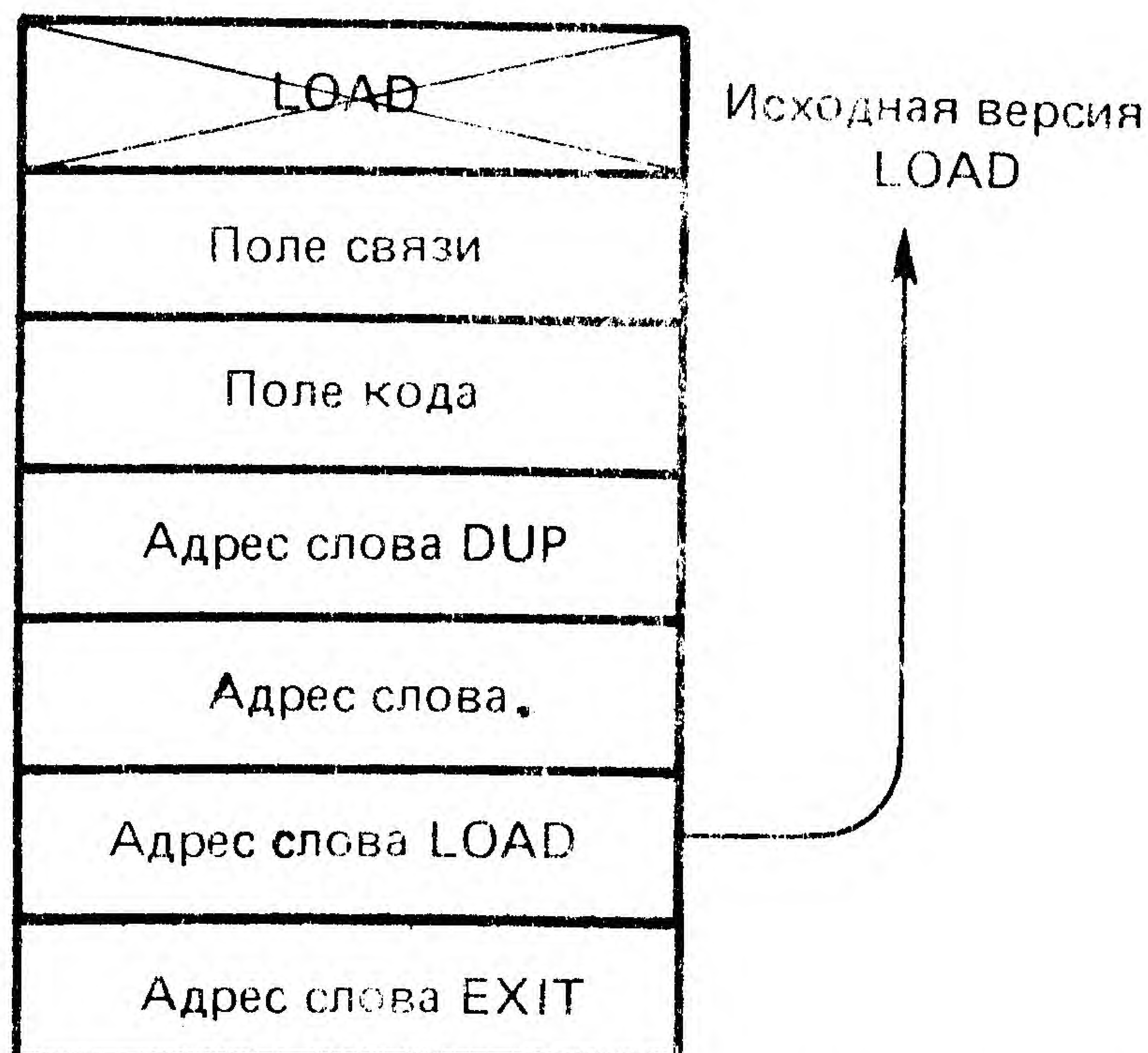
>BODY	(cfa -- pfa)	Вычисление адреса поля параметров определения, "адрес компиляции " которого находится на стеке.
EXIT	(--)	Удаление адреса возврата из вершины стека возвратов и занесение его в указатель адресного интерпретатора. Если слово EXIT скомпилировано в определении через двоеточие, то оно завершает выполнение этого определения в данной точке.
QUIT	(--)	Очистка стека возвратов и передача управления терминалу, ожидающему ввода. Сообщения при этом не выдаются.
ABORT	(--)	Очистка стека данных и выполнение функций слова QUIT. Сообщения не выдаются.

РЕКУРСИЯ

Рекурсией называется обращение процедуры к самой себе. Обычными средствами в Форте запрещается это делать. Например, при вводе

```
: LOAD  ( n -- )  DUP .  LOAD ;
```

вы определяете новую версию слова **LOAD**, которое выдает номер загружаемого блока, после чего обращается к исходному варианту **LOAD**. Чтобы выражения такого типа были возможны, Форт-система во время компиляции определения умышленно «прячет» имя данного определения, и тем самым в новое определение компилируется адрес исходного варианта.



Тем не менее на Форте реализовать рекурсию довольно просто (нехитрые средства защиты Форта легко обмануть). Слово **RECURSE** осуществляет компиляцию адреса текущего определяемого слова. (В некоторых системах это слово известно как **MYSELF**.)[1] Приведем пример рекурсии:

```
VARIABLE СЧЕТЧИК
: ПОЧЕМУ   CR ." Почему вы спрашиваете? "  -1 СЧЕТЧИК +!
   СЧЕТЧИК @ IF  RECURSE  THEN ;
5 СЧЕТЧИК !
```

Выполнив слово ПОЧЕМУ, вы получите:

```
ПОЧЕМУ
Почему вы спрашиваете?
Почему вы спрашиваете?
Почему вы спрашиваете?
Почему вы спрашиваете?
Почему вы спрашиваете? ok
```

---

[1] *Для пользователей систем фиг-Форта.* Определение следующее:

```
: RECURSE   LATEST  PFA CFA , ; IMMEDIATE
```

Для более ранних систем полиФорта:

```
: RECURSE   LAST @ @ 2+ , ; IMMEDIATE
```

Обратите внимание на то, что в рассматриваемом примере нет ни одного цикла. Начиная со значения 5 счетчика, слово ПОЧЕМУ будет обращаться к самому себе до тех пор, пока значение счетчика не станет равным нулю. (Не выполняйте слово ПОЧЕМУ при нулевом или слишком большом значении счетчика, поскольку ваш стек возвратов при этом переполнится.)

ГЕОГРАФИЯ ФОРТА

Вы видите на рисунке *карту памяти*[1] типичной Форт-системы



для одного пользователя. Мультипрограммные системы, такие, как полиФорт, устроены намного сложнее, о чем пойдет речь позднее. А пока рассмотрим простой случай и последовательно изучим каждый район нашей карты.

Предварительно скомпилированное ядро Форта. В памяти с младшими адресами расположен единственный предварительно скомпилированный участок системы (уже скомпилированный в словарную форму). В одних системах коды этого участка хранятся на диске (как правило, блоки 1—8) и автоматически загружаются в память с произвольной выборкой во время запуска или восстановления вашего компьютера, в других — такой участок неизменно находится в программируемой постоянной памяти и становится доступным сразу, как только вы включаете компьютер.

В предварительно скомпилированном участке обычно хранится большая часть математических операций и слов форматизации чисел одинарной длины, операции преобразования стека одинарной длины, команды редактирования, структуры

---

[1] *Для начинающих.* Здесь показано, каким образом распределяется память компьютера в конкретной системе в зависимости от ее назначения. Память разбита на участки по 1024 байта. Это число называется «К» (от слова «кило», означающего тысячу).

управления, ассемблер, все определяющие слова, которые вам уже известны, и, конечно, интерпретаторы текста и адреса[1].

Системные переменные. Следующий раздел памяти содержит *системные переменные*, которые созданы предварительно скомпилированным ядром Форта и используются всей системой. Они, как правило, не применяются пользователем.

Выборочные определения. Тот раздел Форт-системы, который не был предварительно скомпилирован, хранится на диске в виде исходного текста. Какую часть определений из этого раздела загружать, а какую — нет, вы можете решить сами, что улучшит управление использованием памяти вашего компьютера. Блок загрузки для всех выборочных определений называется *блоком выбора*.

Словарь пользователя. Словарь расширяется в сторону увеличения адресов памяти по мере того, как вы добавляете ваши собственные определения в область памяти, называемую *словарем пользователя*. Его следующая доступная ячейка в любой момент времени определяется содержимым переменной с именем **H** (или **DP**). Во время компиляции указатель **H**, по мере того как очередной элемент добавляется к словарю, переходит с ячейки на ячейку (или с байта на байт). Таким образом, для компилятора указатель **H** выступает в качестве закладки; он указывает то место в словаре, куда компилятор может компилировать следующий объект. Этот указатель также используется словом **ALLOT**, которое передвигает его на заданное число байтов. Например, выражение 10 ALLOT добавляет к нему 10 и, следовательно, компилятор зарезервирует память в словаре для массива, состоящего из 10 байтов (или пяти ячеек).

Родственным словом является и **HERE**, которое определяется весьма просто:

```
: HERE  ( -- текущий-адрес )  H @ ;
```

Оно помещает значение **H** в стек. Слово , (запятая) помещает значение одинарной длины в следующую доступную ячейку словаря:

```
: ,  ( n -- )  HERE !  2 ALLOT ;
```

т. е. запоминает некоторое значение в **HERE** и продвигает указатель словаря на два байта, закрепляя память под это значение. С помощью **HERE** вы можете определить, какой объем памяти требуется для любого фрагмента вашей программы, для чего нужно сравнить **HERE** до компиляции и после нее. Например, выражение

```
HERE  220 LOAD  HERE SWAP - . 196 ok
```

---

[1] *Для специалистов.* Чтобы получить представление о том, как компактен Форт, вам достаточно знать, что весь предварительно скомпилированный участок поли-Форта занимает меньше 8К байтов памяти.

показывает, что определения, загруженные блоком 220, заняли 196 байтов памяти словаря.

Рабочая область (PAD). На некотором удалении от **HERE** вы обнаружите в своем словаре небольшую область памяти, называемую *рабочей областью.* Наша рабочая область обычно служит для хранения строк символов в коде ASCII, которые подвергаются обработке перед выводом на терминал. Так, слова, осуществляющие форматирование чисел, используют рабочую область для обработки чисел в коде ASCII во время их перевода, прежде чем вывести эти числа с помощью **TYPE.**

Размер рабочей области не определен.

В большинстве систем расстояние между началом рабочей области и вершиной стека данных измеряется сотнями и даже тысячами байтов.

Поскольку адрес начала рабочей области определяется относительно последнего элемента словаря, он изменяется всякий раз при добавлении нового определения либо выполнении **FORGET** или **EMPTY**.

Подобная организация тем не менее гарантирует безопасность, так как рабочая область никогда не используется при выполнении перечисленных выше действий. Слово **PAD** вносит в вершину стека текущий адрес начала рабочей области. Оно имеет довольно простое определение:

```
: PAD  ( -- a)  HERE  34 + ;
```

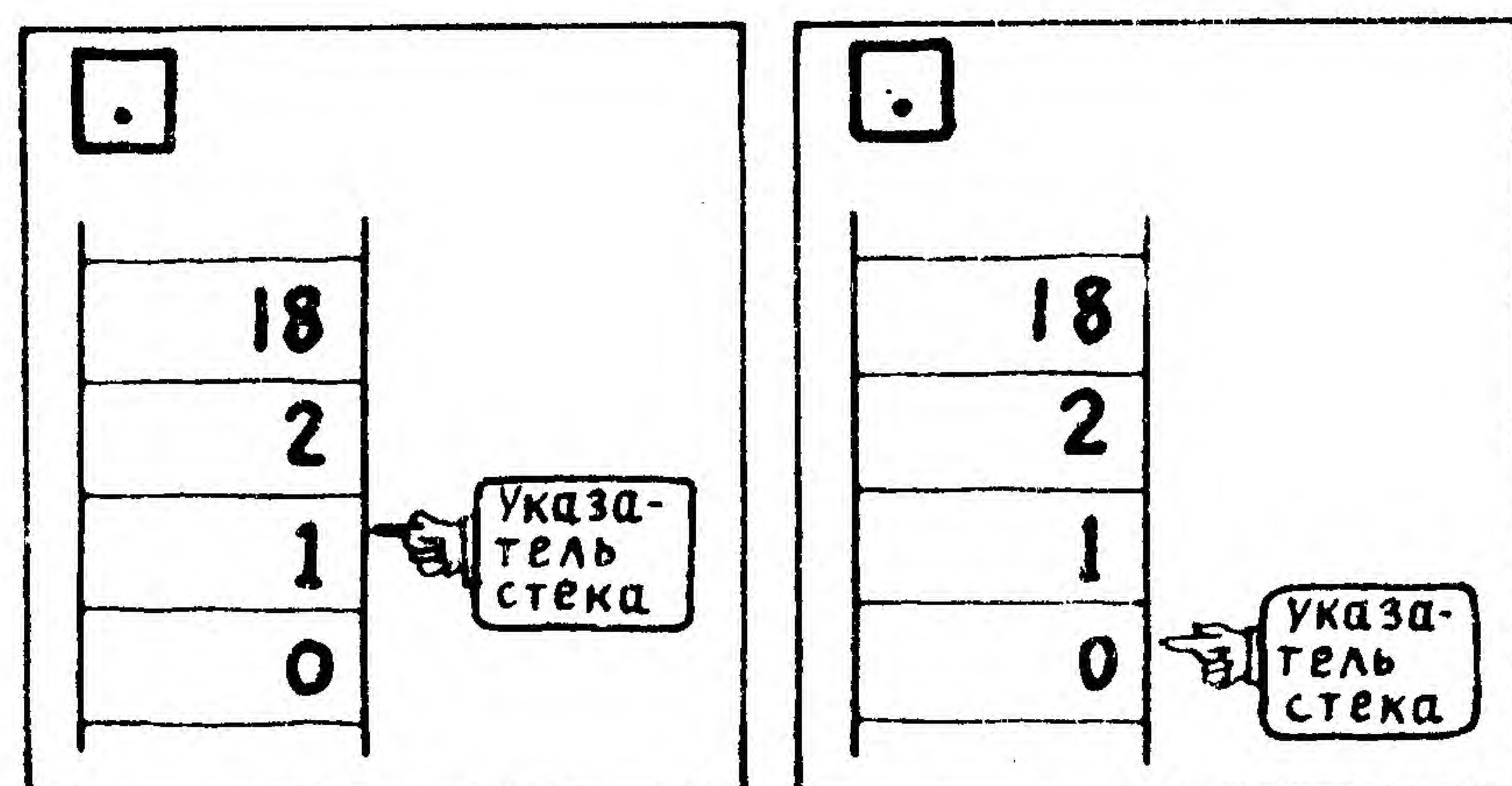
т. е. помещает в стек адрес, который отстоит на фиксированное число байтов от **HERE** (в действительности это число может меняться).

Стек данных. Намного выше рабочей области расположен участок, зарезервированный под стек данных. У вас может создаться впечатление, что значения где-то передвигаются вверх и вниз, как будто их кто-то «вталкивает» и «выталкивает», однако на самом деле ничего подобного не происходит. Единственное, что изменяется, — это указатель вершины стека.

Как вы увидите в дальнейшем, при занесении некоторого числа в стек фактически лишь уменьшается указатель (т. е. указывает на следующий участок в направлении к младшим адресам памяти), а затем число запоминается в том месте, куда показывает указатель. Когда вы удаляете число из стека, оно выбирается из участка, на который показывает указатель, после чего последний увеличивается.

Все числа, расположенные на карте памяти выше указателя стека, не имеют смысла.





По мере добавления в стек новых значений он «растет» в направлении младших адресов памяти.

Указатель стека выбирается посредством слова **SP@** [1]. Так как это слово доставляет адрес самого верхнего участка стека, выражение **SP@** выбирает содержимое вершины стека. Такая операция, конечно, идентична операции **DUP**. Если у вас в стеке находится пять значений, то пятое значение можно скопировать с помощью выражения:

```
SP@ 8 + @
```

(но в большинстве случаев это не может считаться хорошим стилем программирования).

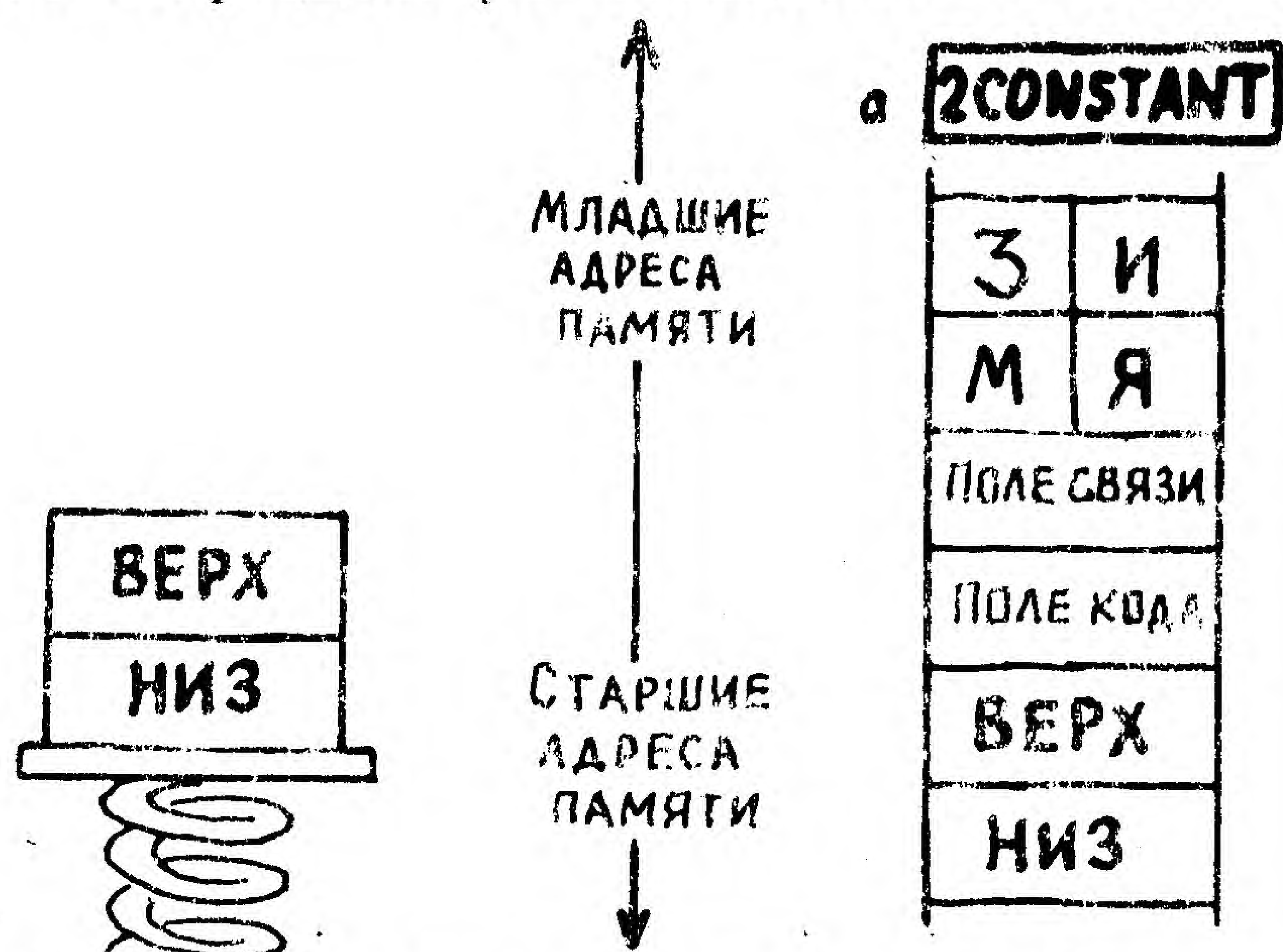
На дно стека указывает переменная **SO**. Она всегда содержит адрес ячейки, расположенной непосредственно *под* ячейкой, соответствующей «пустому» стеку.



Заметим, что при размещении чисел двойной длины как в стеке, так и в словаре старшая по порядку ячейка размещается по

[1] *Для пользователей систем полиФорта.* Это слово имеет имя 'S.

младшему адресу. При выполнении операций **2!** и **2@** (см. рисунок). Этот порядок размещения ячеек сохраняется.



Буфер входного текста. Буфер входного текста представляет собой область памяти длиной, как правило, 80 байт, куда поступают вводимые с клавиатуры символы после нажатия клавиши «возврат каретки». Именно здесь они будут просмотрены текстовым интерпретатором.

Не путайте этот термин с термином *входной поток* (см. гл. 3), который означает последовательность слов, подлежащих интерпретации. Данная последовательность может быть расположена либо в буфере входного текста (в режиме интерпретации), либо в блоке, содержащем исходный текст (в режиме загрузки).

Буфер входного текста увеличивается в направлении старших адресов памяти (в том же направлении, что и рабочая область (PAD)). Слово **TIB** выбирает начальный адрес буфера. (На фиг-Форте вы можете ввести "TIB @ ", на полиФорте — "SO @ ".)

Стек возвратов. Выше буфера входного текста расположен стек возвратов, функционирование которого идентично функционированию стека данных.

Пользовательские переменные. Следующий раздел памяти содержит *пользовательские переменные*. Эти переменные включают в себя слова **BASE**, **SO** и многие другие, которые мы рассмотрели ниже.

Блочные буферы. В самой верхней области памяти расположены буферы блоков. Каждый буфер имеет объем 1024 байта для размещения содержимого дискового блока. Всякий раз, когда вы осуществляете доступ к какому-либо блоку (например, распечатывая или загружая его), система копирует данный блок с диска в такой буфер, где он может изменяться с помощью редактора или интерпретироваться посредством слова **LOAD**. Более подробно мы обсудим блочные буферы в гл. 10.

На этом наше путешествие по карте памяти типичной Форт-системы индивидуального пользования завершается. Далее приводится перечень уже знакомых вам слов, которые используются при работе с различными участками памяти.

H или DP	(-- a)	Занесение в стек адреса указателя словаря.
HERE	(-- a)	Занесение в стек адреса очередного доступного участка словаря.
PAD	(-- a)	Занесение в стек адреса начала рабочей области, в которой хранятся строки символов в процессе промежуточной обработки.
SP@ или 'S	(-- a)	Занесение в стек адреса вершины стека данных до того, как исполнено само слово SP@.
S0	(-- a)	Содержит адрес дна стека данных.
TIB	(-- a)	Занесение в стек адреса начала буфера входного текста.

МУЛЬТИЗАДАЧНЫЕ ФОРТ-СИСТЕМЫ

Наряду с однозадачными существуют и мультизадачные Форт-системы[1]. Они могут работать с произвольным числом задач. Задача может быть либо *терминальной*, при выполнении которой вся интерактивная мощь Форта передается оператору, сидящему за терминалом, либо управляющей, которая обеспечивает управление аппаратным средством, не имеющим терминала.

Любой из задач нужна своя *пользовательская область*. Размер и содержимое пользовательской области зависят от вида задачи. Типовые структуры для двух видов задач показаны на рисунке.

Каждой терминальной задаче требуется собственный словарь, рабочая область (PAD), стек данных, буфер входного текста, стек возвратов и пользовательские переменные. Это означает, что все определяемые вами слова, как правило, *недоступны* другим задачам. Кроме того, все задачи имеют свои собственные копии пользовательских переменных, таких, как **BASE**.

Управляющая задача имеет пару стеков и небольшой набор пользовательских переменных. Так как при выполнении управляющей задачи не используется терминал, ей не требуются ни собственный словарь, ни рабочая область, ни буфер входного текста.

ПОЛЬЗОВАТЕЛЬСКИЕ ПЕРЕМЕННЫЕ

Пользовательская переменная в отличие от обычной (определяемой с помощью слова **VARIABLE**), значение которой хранится в поле параметров элемента словаря, состоит из двух частей. Сами данные хранятся в массиве, называемом *пользовательской таблицей*. Элемент словаря для каждой пользовательской переменной расположен в другом месте; он содержит смещение в пользовательской таблице. Когда вы выполняете имя пользовательской пере-

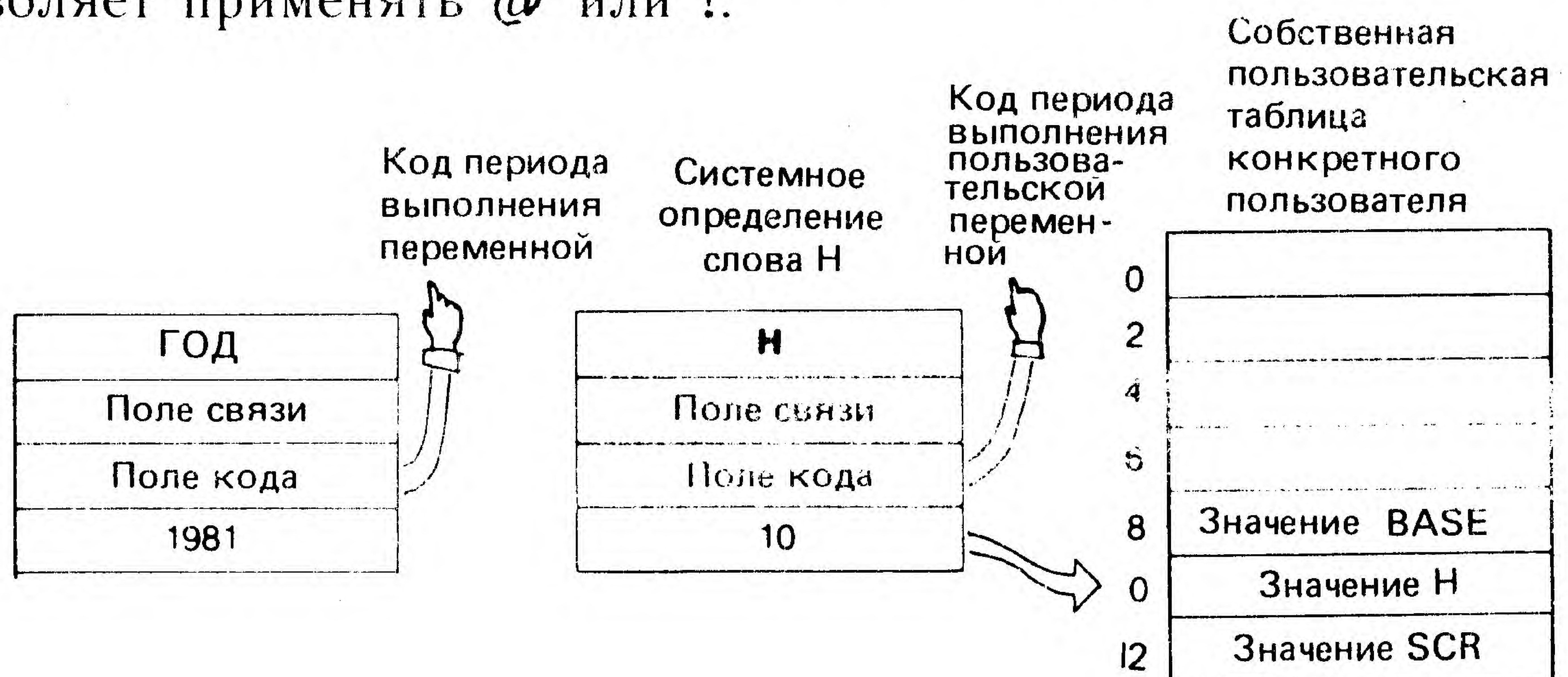
[1] *Для начинающих.* Термин «мультизадачная» описывает систему, в которой многочисленные задачи выполняются одновременно на одном и том же компьютере, не оказывая влияния друг на друга.

ПОЛЬЗОВАТЕЛЬСКИЕ ПЕРЕМЕННЫЕ
СТЕК ВОЗВРАТОВ
СТЕК ДАННЫХ

ПОЛЬЗОВАТЕЛЬСКИЕ ПЕРЕМЕННЫЕ
СТЕК ВОЗВРАТОВ
БУФЕР ВХОДНОГО ТЕКСТА
СТЕК ДАННЫХ
РАБОЧАЯ ОБЛАСТЬ (pad)
СЛОВАРЬ ПОЛЬЗОВАТЕЛЯ

Блочные буферы
Задача, получаемая в результате первоначальной загрузки системы, — OPERATOR
ПОЛЬЗОВАТЕЛЬСКАЯ ОБЛАСТЬ 4 (Управляющая задача)
ПОЛЬЗОВАТЕЛЬСКАЯ ОБЛАСТЬ 3 (терминальная задача)
ПОЛЬЗОВАТЕЛЬСКАЯ ОБЛАСТЬ 2 (терминальная задача)
ПОЛЬЗОВАТЕЛЬСКАЯ ОБЛАСТЬ 1 (терминальная задача)
ВЫБОРОЧНЫЕ ОПРЕДЕЛЕНИЯ
СИСТЕМНЫЕ ПЕРЕМЕННЫЕ
ПРЕДВАРИТЕЛЬНО СКОМПИЛИРОВАННАЯ ОБЛАСТЬ ФОРТА
СТАРШИЕ АДРЕСА ПАМЯТИ
МЛАДШИЕ АДРЕСА ПАМЯТИ

менной, например **Н**, смещение добавляется к начальному адресу пользовательской таблицы, что дает вам адрес **Н** в этом массиве и позволяет применять @ или !.



Основное достоинство пользовательских переменных состоит в том, что любое число задач может использовать одно и то же *определение* некоторой переменной, и каждая задача может получать свое собственное *значение* этой переменной. Всякая задача, которая выполняет выражение

```
BASE @
```

получает значение **BASE** из своей пользовательской таблицы, благодаря чему экономится большой объем памяти, и тем не менее она может выполняться независимо от остальных задач.

Порядок размещения пользовательских переменных в таблице и значения их смещений изменяются от системы к системе.

Итак, существуют переменные трех видов:

- системные, применяемые всей Форт-системой;
- пользовательские, являющиеся уникальными для каждой задачи, несмотря на то что их определения могут быть использованы всеми задачами системы;
- обычные, которые могут быть доступными либо во всей системе, либо в пределах единственной задачи (в зависимости от того, как они определены: внутри слова **OPERATOR** или внутри частной задачи).

КОНТЕКСТНЫЕ СЛОВАРИ (СПИСКИ СЛОВ)

В простой Форт-системе имеются три штатных контекстных словаря: словарь Форта, словарь редактора и словарь ассемблера.

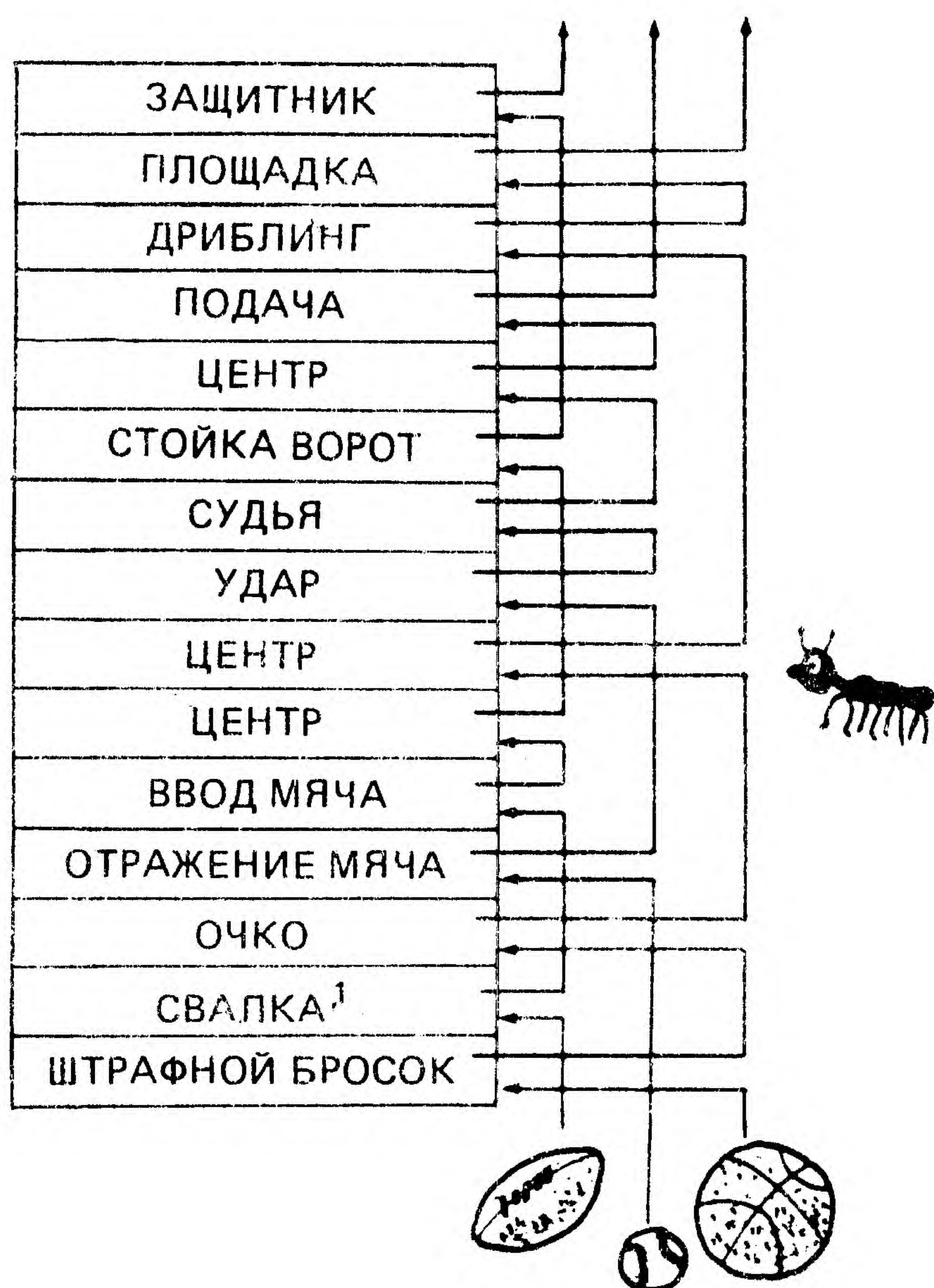
Все рассмотренные выше слова принадлежат словарю Форта, за исключением команд редактора, которые относятся к словарю редактора. В словарь ассемблера включены команды, предназначенные для программирования на языке Ассемблера конкретного компьютера.

Определения добавляются в один и тот же словарь в порядке их компиляции, независимо от того, к какому контекстному словарю они принадлежат. Таким образом, контекстные словари являются не подразделами словаря как участка памяти, а связанными списками, переплетенными между собой внутри этого общего словаря[1].

В качестве примера рассмотрим три контекстных словаря: «футбол», «бейсбол» и «баскетбол» (см. рисунок). Все они совместно существуют в одном и том же общем словаре, однако апостроф, проходя по цепи, имеющей отношение, скажем, к баскетболу, пере-

---

[1] Необходимо отличать общий словарь (dictionary) как прерывный участок памяти для размещения слов от словаря (vocabulary) — связанного списка слов. Примерами последнего могут служить словари Форта, редактора и ассемблера. — *Примеч. ред.*



бирает только слова из баскетбольного словаря. Даже если в каждом контекстном словаре есть слово ЦЕНТР, апостроф подберет его вариант, требуемый для данного контекста.

Помимо замкнутости контекстных словарей, необходимо отметить еще одно преимущество такой организации данных — скорость поиска. Если в нашем примере идет речь о баскетболе, то зачем нам перебирать слова, относящиеся к футболу и бейсболу?

Вы можете переключить контекст поиска по словарю, выполнив любую из трех команд: **FORTH**, **EDITOR** или **ASSEMBLER**. Например, вводя слово **FORTH**, вы уверены, что поиск будет осуществляться в контексте словаря Форта. Но, как правило, Форт-система изменяет для вас контекст автоматически. Рассмотрим типичную схему. Система начинает работу с контекста словаря Форта. Допустим, вы заносите некоторую программу в блок. Конкретные команды редактора переключают контекст на словарь редактора. Вы работаете в контексте словаря редактора до тех пор, пока не осуществите загрузку вашего блока и не приступите к компиляции определений. Слово : автоматически восстанавливает контекст, который был ранее, а именно: Форт.

Различные версии Форт-систем имеют различные реализации контекстных словарей. Однако существуют некоторые общие положения, которые можно распространить на большинство систем.

Словарь, где должен осуществляться поиск, определяется пользовательской переменной с именем **CONTEXT** (КОНТЕКСТ). Как

уже упоминалось выше, команды **FORTH**, **EDITOR** и **ASSEMBLER** изменяют контекст поиска.

Известен еще один вид контекста: словарь, к которому должно быть присоединено новое определение. Такой словарь задается другой переменной с именем **CURRENT** (ТЕКУЩИЙ). Поскольку **CURRENT** обычно определяет словарь Форта, то новые определения, как правило, присоединяются к данному словарю.

А каким образом система компилирует слова в словари редактора и ассемблера? Это делается с помощью слова **DEFINITIONS** (ОПРЕДЕЛЕНИЯ), например:

```
EDITOR DEFINITIONS
```

Вы знаете, что слово **EDITOR** устанавливает **CONTEXT** для редактора. Слово **DEFINITIONS** копирует содержимое слова **CONTEXT**, каким бы оно ни было, в слово **CURRENT**. Слово **DEFINITIONS** имеет простое определение:

```
: DEFINITIONS  CONTEXT @  CURRENT ! ;
```

После вывода выражения EDITOR DEFINITIONS все компилируемые с этого момента слова заносятся в словарь редактора до тех пор, пока вы не введете выражение FORTH DEFINITIONS, чтобы поместить в **CURRENT** Форт.

Техника работы с контекстными словарями существенно зависит от конкретной системы и в какой-то степени противоречива. В Стандарте-83 подробности опущены, а потому обойдемся без них и мы. Обращайтесь к документации по своей системе.

' xxx	(-- a)	Осуществляется поиск в словаре адреса слова xxx (следующего слова из входного потока).
[']	период-компиляции: (--) период-выполнения: (--a)	Используется только в определении через двоеточие. Компиляция адреса следующего слова из определения как литерала.
EXECUTE	(a --)	Выполнение элемента словаря, адрес поля параметров которого находится на стеке.
@EXECUTE	(a --)	Выполнение элемента словаря, на pfa которого ссылается содержимое по адресу a. Если адрес содержит нуль, то @EXECUE ничего не выполняет.
>BODY	(cfa -- pfa)	Вычисление адреса поля параметров определения, адрес компиляции которого находится на стеке.
EXIT	(--)	Удаление адреса возврата из вершины стека возвратов и занесение его в указатель адресного интерпретатора. Если слово EXIT скомпилировано в определении через двоеточие, то оно завершает выполнение этого определения в данной точке.

QUIT	(--)	Очистка стека возвратов и передача управления терминалу, ожидающему ввода. Сообщения при этом не выдаются.
ABORT	(--)	Очистка стека данных и выполнение функций слова QUIT. Сообщения не выдаются.
H или DP	(-- a)	Занесение в стек адреса указателя словаря.
HERE	(-- a)	Занесение в стек адреса очередного доступного участка словаря.
PAD	(-- a)	Занесение в стек адреса начала рабочей области, в которой хранятся строки символов в процессе промежуточной обработки.
SP@ или 'S	(-- a)	Занесение в стек адреса вершины стека данных до того, как исполнено само слово SP@.
S0	(-- a)	Содержится адрес дна стека данных.
TIB	(-- a)	Занесение в стек адреса начала буфера входного текста.

ОСНОВНЫЕ ТЕРМИНЫ

Адресный интерпретатор. Второй из двух интерпретаторов Форта выполняет список адресов из элемента словаря, являющегося определением через двоеточие.

Буфер входного текста. Участок памяти внутри терминальной задачи, используемый для хранения текста по мере его поступления с терминала. Именно здесь интерпретируется поступивший исходный текст.

Выборочные определения. Набор определений Форта, которые входят в систему, но хранятся отдельно от предварительно сгенерированных определений. Так называемый «блок выбора» загружает блоки, содержащие выборочные определения. Этот блок по желанию пользователя может быть изменен.

Загрузка. Загрузка предварительно скомпилированного участка Форта в компьютер, после которой вы можете работать на Форте. Она происходит автоматически, когда вы включаете компьютер или нажимаете клавишу «Восстановление».

Заголовок. Поля имени и связи в элементе словаря Форта.

cfa. Адрес поля кода; адрес поля кода в элементе словаря.

Задача. Применительно к Форту это участок памяти, содержащий как минимум параметр и стек возвратов, а также множество пользовательских переменных.

Код периода выполнения. Программа, которая хранится в памяти в скомпилированном виде и определяет, что происходит во время выполнения слов заданного класса. Код периода выполне-

ния для определения через двоеточие осуществляет переход на следующий уровень и передает управление адресному интерпретатору. В случае переменной код периода выполнения помещает содержимое pfa данной переменной в вершину стека.

Контекстный словарь. Независимо связанное подмножество словаря Форта.

Векторные вычисления. Способ вычислений, при котором выполняемый код задается не своим адресом, а адресом некоторого участка, где содержится адрес этого кода. Такой участок обычно называют «вектором». При изменении состояния системы вектор может быть переключен на некоторый другой участок кода.

Определяющее слово. Слово Форта, создающее элемент словаря, например : **CONSTANT, VARIABLE** и т. д.

Поле кода. Ячейка в элементе словаря, содержащая адрес кода периода исполнения для данного конкретного типа определения.

Поле имени. Участок элемента словаря, содержащий имя (или сокращение этого имени) определенного слова, а также число символов в данном имени.

Поле связи. Ячейка элемента словаря, содержащая адрес предыдущего определения; применяется при поиске по словарю (в системах, использующих несколько цепочек, поле связи содержит адрес предыдущего определения в этой же цепочке).

Поле параметров. Участок элемента словаря, где находится «содержимое» данного определения: в случае **CONSTANT** — значение константы, в случае **VARIABLE** — значение переменной, в случае определения через двоеточие — список адресов слов, которые должны выполняться поочередно во время выполнения этого определения. Длина поля параметров меняется в зависимости от назначения.

Пользовательская переменная. Переменная Форта, к которой в отличие от пользовательской переменной возможен доступ из любой задачи системы.

pfa. Адрес поля параметров; адрес первой ячейки поля параметров элемента словаря (или, если поле параметров представляет собой одну ячейку, адрес последней).

Рабочая область (PAD). Участок памяти внутри терминальной задачи, который служит рабочей областью для хранения строк символов в процессе промежуточной обработки.

Системная переменная. Переменная Форта, которая в отличие от пользовательской переменной доступна по всей системе (в любой задаче).

Ссылка вперед. Ссылка на слово, которое еще не определено. В Форте — это способ реализации векторных вычислений.

Тело элемента. Поля кода и параметров элемента словаря Форта.

Терминальная задача. Задача в мультизадачной системе, обеспечивающая взаимодействие с оператором через терминал, т. е. задача, имеющая текстовый интерпретатор, словарь и т. д.

Управляющая задача. В мультизадачной системе — задача, которая не допускает взаимодействия с терминалом. Управляющие задачи, как правило, обслуживают аппаратные средства.

Участок предварительно скомпилированных определений. Часть Форт-системы, которая постоянно присутствует в памяти в объектном коде сразу после выполнения операции начальной загрузки или восстановления. Эта часть обычно включает в себя текстовый и адресный интерпретаторы, определяющие слова, структуры управления, операции одинарной длины и операции над стеком, команды преобразования чисел одинарной длины и их форматизирования, редактор, ассемблер.

УПРАЖНЕНИЯ

9.1. Напишите определение слова ПОЛУЧАЕМ так, чтобы его функции могли корректироваться словами СКЛАДЫВАЯ и УМНОЖАЯ, как в приведенном ниже примере:

```
СКЛАДЫВАЯ 2 3 ПОЛУЧАЕМ 5 ok
УМНОЖАЯ 2 3 ПОЛУЧАЕМ 6 ok
```

9.2. Каков начальный адрес вашего личного словаря?

9.3. Как далеко расположена рабочая область (PAD) в вашей системе от конца вашего личного словаря?

9.4. Если слово ДАТА определено через **VARIABLE**, то в чем состоит различие между следующими фразами: ДАТА . и ' ДАТА >BODY .? А чем отличаются друг от друга эти две фразы:
BASE . и ' BASE >BODY .?

9.5. В этом упражнении вы должны создать массив для векторных (косвенных) вычислений, т. е. массив с адресами слов Форта. Определите одномерный массив ячеек, содержащих по два байта каждая, который будет возвращать адрес n-го элемента при заданном индексе n. Определите несколько слов так, чтобы они обеспечивали вывод на ваш дисплей некоторой информации и ничего не вводили. Запомните адреса этих слов в различных элементах массива, а адрес ничего не выполняющего слова — в остальных его элементах. Определите слово, вырабатывающее правильный индекс и выполняющее слово, адрес которого расположен в соответствующем индексу элементе, например:

```
0 ВЫПОЛНИ Привет, я говорю на форте. ok
1 ВЫПОЛНИ 1 2 3 4 5 6 7 8 9 10 ok
2 ВЫПОЛНИ
**********
**********
**********
**********
**********
3 ВЫПОЛНИ ok
4 ВЫПОЛНИ ok
```

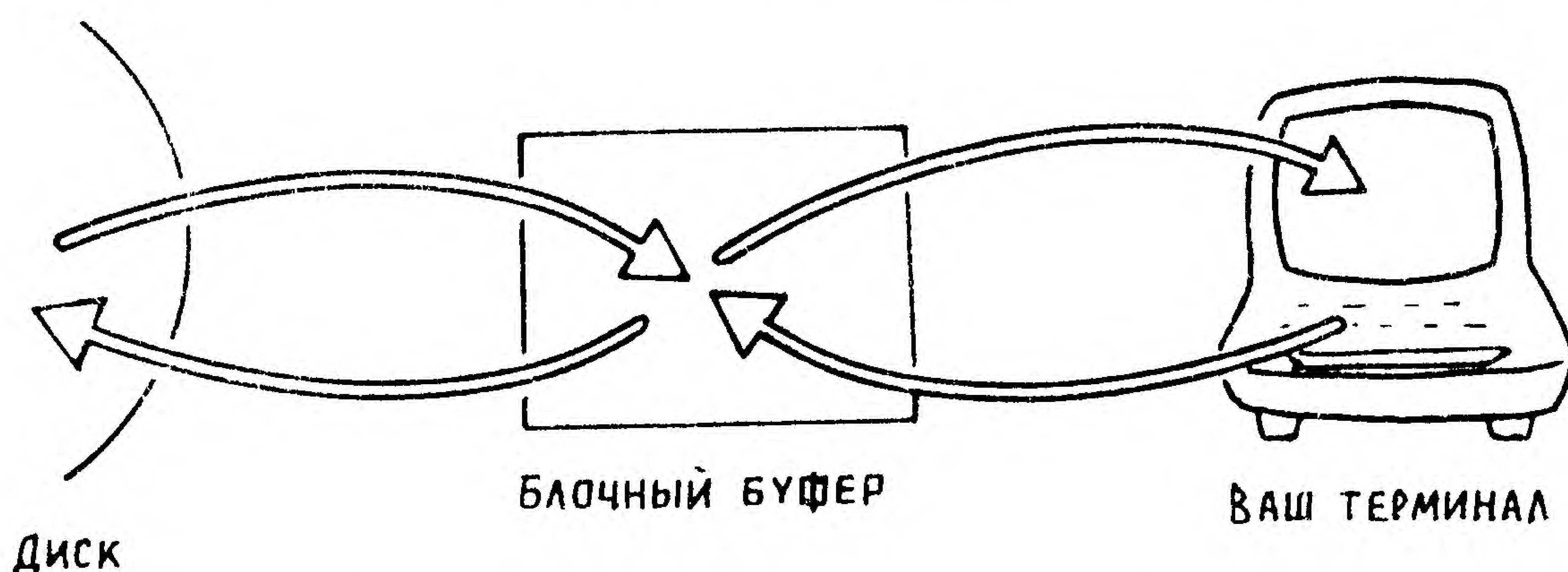
Глава 10
ВВОД-ВЫВОД

В настоящей главе мы объясним вам, как Форт-система управляет вводом-выводом, как текст поступает с клавиатуры и выводится на экран дисплея, как данные записываются во внешнюю память и считываются из нее. Особое внимание будет уделено командам доступа к дискам, вывода, работы со строками, ввода и преобразования вводимых чисел.

БЛОЧНЫЕ БУФЕРЫ

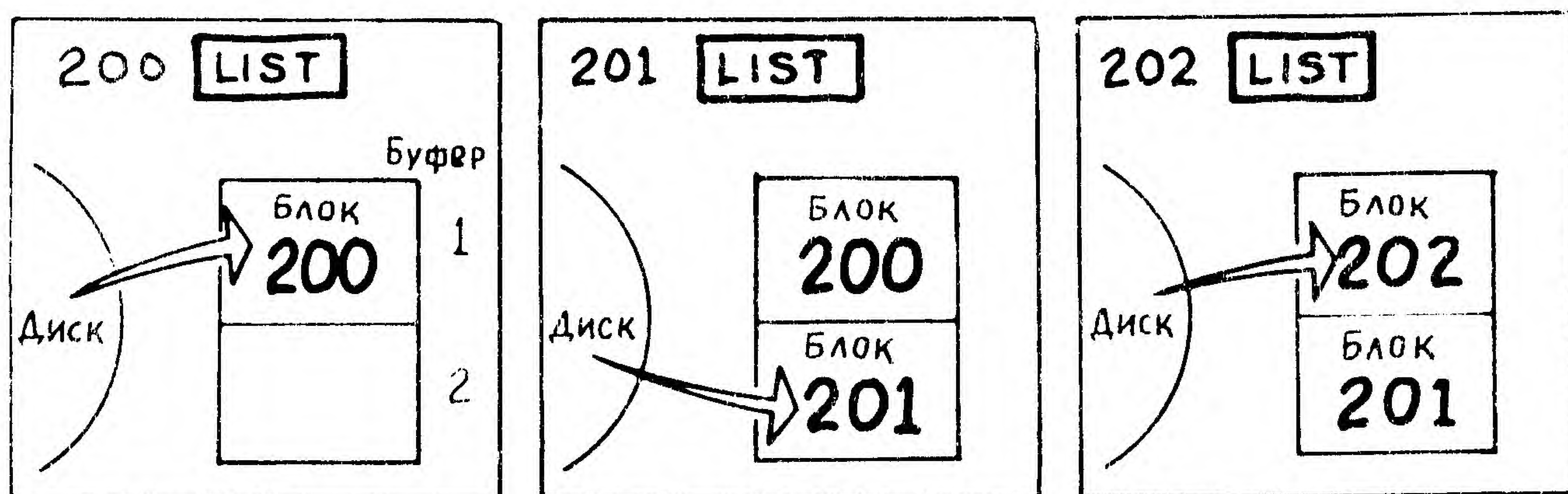
Форт-система построена таким образом, чтобы вы не задумывались о том, как организованы блочные буферы. Однако ничего сложного в этом нет, так что вы всегда сможете в них разобраться, применить и (при необходимости) исправить их. Поэтому мы опишем здесь механизм работы блочных буферов.

Как уже упоминалось ранее, каждый буфер имеет достаточные размеры для размещения содержимого одного блока (1024 байта) в памяти с произвольной выборкой (ОЗУ), поэтому он может редактироваться и загружаться или к нему просто может быть осуществлен доступ любыми иными средствами. Мы думаем, что непосредственно взаимодействуем с диском, хотя на самом деле система перекачивает данные с диска в буфер, откуда их можно считывать. Можно также записать данные в буфер с тем, чтобы система переслала их дальше, на диск.



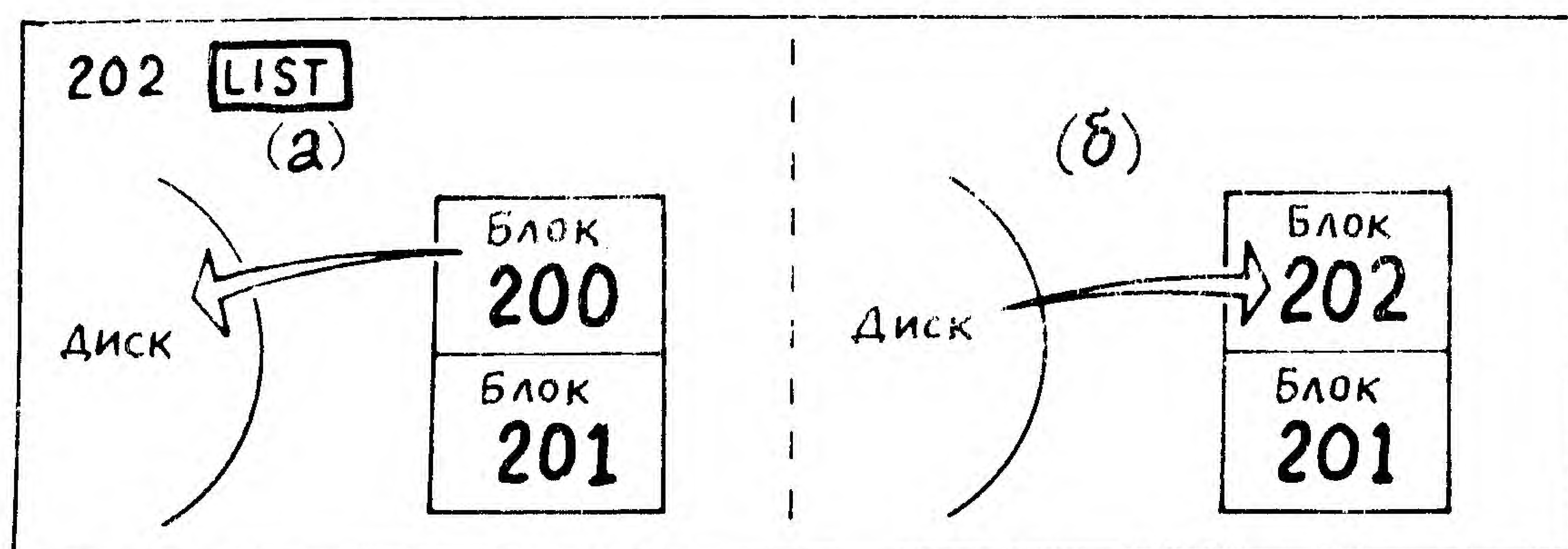
Здесь мы имеем дело с так называемой *виртуальной памятью,* т. е. работаем с памятью большой емкости, как с памятью компьютера.

Во многих Форт-системах (даже если они являются мультипрограммными) используются всего лишь два блочных буфера. Давайте выясним, почему это возможно.



Допустим, в вашей системе имеются два буфера и вы выполняете следующие действия. Сначала вы распечатываете блок 200. Система считывает его с диска и пересылает в буфер 0, откуда слово **LIST** выводит этот блок на печать. Далее вы распечатываете блок 201. Система копирует блок 201 с диска в другой буфер. И наконец, вы распечатываете блок 202. Система копирует данный блок с диска в буфер, который применялся в первую очередь, а именно в буфер 0.

Что случилось с бывшим содержимым буфера 0? Оно просто было замещено (затерто) новым содержимым. Старая информация для нас не потеряна, так как блок 200 все еще остается на диске. Но если бы вы *отредактировали* блок 200, были бы утеряны ваши исправления? Нет, конечно. При распечатке блока 202 *сначала* измененное содержимое блока 200 посылается на диск с тем, чтобы обновить прежнее содержимое этого блока, а *затем* в буфер помещается содержимое блока 202.



Волшебное слово **UPDATE** устанавливает (обновляет) флаг, связанный с каждым буфером. Этот флаг указывает, что содержимое текущего блока (к которому осуществлялся самый последний доступ) после прочтения с диска подвергалось изменениям. Все команды редактора, изменяющие содержимое блоков, — будь то занесение или удаление — имеют слово **UPDATE** в своих определениях. Если в некоторый буфер требуется поместить содержимое

другого блока и флаг обновления данного буфера установлен, прежнее содержимое не исчезает, а копируется снова на диск.

Основным словом, которое считывает содержимое какого-либо блока с диска, является **BLOCK**. Например, если вы введете

```
205 BLOCK
```

то система скопирует блок 205 в один из буферов. Кроме того, **BLOCK** оставляет в вершине стека адрес начала этого буфера. Используя адрес начала буфера как базу, вы можете получить доступ к любому байту блока[1]. Слово **BLOCK** получает управление всякий раз, когда вы распечатываете содержимое блока посредством **LIST** или загружаете его с помощью **LOAD**.

Рассмотрим выполнение слова **BLOCK** более подробно. В первую очередь **BLOCK** проверяет, не находится ли уже содержимое нужного блока в некотором буфере, и если оно там находится, то помещает в вершину стека адрес данного буфера, а если нет — выбирает буфер (в большинстве систем буфер с самым давним доступом). В том случае, когда буфер подвергался изменениям, система копирует его содержимое на диск, а затем копирует содержимое нужного блока в этот буфер.

Такая организация позволяет многократно модифицировать содержимое блока, не обращаясь всякий раз к дисководу. Поскольку обращение к диску занимает больше времени, чем к памяти с произвольной выборкой, при этом экономится масса времени. С другой стороны, когда в одной системе работают несколько пользователей, подобная организация дает им возможность обходиться всего лишь двумя буферами (2К памяти) даже в том случае, если каждый из них осуществляет доступ к различным блокам.

Многие Форт-системы предоставляют своим пользователям право задавать число блочных буферов, обеспечивая им тем самым возможность выбора между размером доступной оперативной памяти и частотой обращения к дискам.

Слово **FLUSH** (ВЫБРОС) инициирует немедленную запись всех обновленных буферов на диск. (Так как вы теперь имеете представление о буферах, можно объяснить, для чего нужно слово **FLUSH**: обновление буфера не означает записи его содержимого на диск.) Помимо прочего после употребления слова **FLUSH** система «забывает», что в буферах хранилось содержимое блоков (она освобождает все буферы). Если вам понадобится распечатать или загрузить один из таких блоков, то придется считать его содержимое посредством **BLOCK** с диска снова.

---

[1] *Для пользователей некоторых систем фиг-Форта.* Блоки, занимающие 1024 байта, могут быть считаны в нескольких несмежных буферов, что затрудняет индексирование блоков (см. листинги системы фиг-Форта).

Определенное Стандартом-83 слово **SAVE-BUFFERS** (СОХРАНИТЬ-БУФЕРЫ) по своим возможностям беднее слова **FLUSH**: оно сохраняет на диске содержимое обновленных буферов, но не освобождает их. Если вам снова потребовался некоторый блок, то его содержимое уже находится в одном из буферов, и Форт-система в этом случае *не обращается* к диску.

Как правило, вам нет необходимости пользоваться приведенными выше словами, поскольку слово **BLOCK** гарантирует, что перед повторным применением буферов их содержимое будет сохранено на диске. Однако во время отладки новой программы вам эти слова могут пригодиться (прежде чем система выйдет из строя, исправления все же должны попасть на диск). **SAVE-BUFFERS** переписывает хранящиеся в буферах блоки на диск, но с содержимым буферов можно работать и в дальнейшем. Это сокращает число обращений к диску (что экономит время и позволяет избежать многих неприятностей). Слово **FLUSH** необходимо при смене дисков, так как оно эффективно освобождает буферы от их прежнего содержимого, или в тех случаях, когда вы хотите убедиться в том, что информация *действительно* была записана на диск.

К сожалению, перечисленные слова и их функции изменяются от диалекта к диалекту до неузнаваемости. Ниже приводятся описания этих слов для различных систем.

	FLUSH	SAVE-BUFFERS
Фиг-Форт	Копирование всех обновленных буферов во внешнюю память и их освобождение	(Не определено)
Стандарт-79	(Не определено: переименованное слово SAVE-BUFFERS)	Копирование всех обновленных буферов во внешнюю память и их освобождение
Стандарт-83	Копирование всех обновленных буферов и их освобождение	Копирование всех обновленных буферов во внешнюю память, сброс флагов обновления этих буферов без их освобождения

На всех диалектах слово **EMPTY-BUFFERS** заставляет систему «забыть» о том, что у нее какие-то блоки размещены в буферах и сбрасывает все флаги обновения без записи содержимого блоков на диск. **EMPTY-BUFFERS** полезно применять в тех случаях, если вы случайно испортили[1] содержимое некоторого буфера (на-

[1] *Для начинающих.* Испорченные, бессмысленные или не имеющие отношения к обработке, ради которой были введены, данные программисты называют «мусором».

пример, удалили несколько нужных строк, а их текст забыт или просто что-нибудь напутали) и не хотите, чтобы оно попало на диск. Когда вы после выполнения этого слова снова читаете свой блок, система не выясняет, есть ли содержимое вашего блока в памяти, а просто считывает его с диска[1].

Согласно Стандарту-83 слово **FLUSH** можно определить следующим образом:

```
: FLUSH   SAVE-BUFFERS  EMPTY-BUFFERS ;
```

Слово **BUFFER** заносит информацию на диск без учета прежнего содержимого диска (например, при инициализации диска, записи потока информации, копировании ленты на диск и т. д.). **BUFFER** используется словом **BLOCK** для назначения номера блока следующему доступному буферу. Само слово **BUFFER** содержимое с диска в буфер не считывает (хотя в некоторых системах считывает). К тому же оно не проверяет, был ли номер блока присвоен какому-либо буферу, и может случиться так, что один и тот же номер будут иметь два буфера. Такую ситуацию вы обязаны контролировать сами.

UPDATE	(—)	Блок, доступ к которому осуществлялся в последнюю очередь, отмечается как модифицируемый. Этот блок будет впоследствии переписан во внешнюю память, если его буфер потребуется для размещения другого блока или в результате выполнения слова FLUSH.
SAVE-BUFFERS	(—)	Запись содержимого всех обновленных буферов в соответствующие блоки внешней памяти. У всех буферов погашается признак обновления, но они продолжают оставаться распределенными.
FLUSH	(—)	Осуществляется SAVE-BUFFERS, затем происходит погашение признака обновления всех буферов. Используется при установке или смене накопителей внешней памяти.
EMPTY-BUFFERS	(—)	Все блочные буфера отмечаются как пустые независимо от их содержания. Обновленные блоки во внешнюю память не записываются.

[1] *Для пользователей мультипрограммной системы.* Будьте осторожны! Слово **EMPTY-BUFFERS** освобождает *все* буферы. Программа для работы с базой данных, широко применяющая средства восстановления после ошибок, не должна делать это с помощью **EMPTY-BUFFERS**. Но Форт-программы, использующие блоки, легко расширить с тем, чтобы они удовлетворяли запросы любой пользовательской программы.

BLOCK	(u -- a)	Занесение в стек адреса первого байта в блоке u . Если данного блока еще в памяти нет, то происходит его пересылка из внешней памяти в тот буфер, к которому дольше всех не было доступа. Если блок, занимающий данный буфер, обновлялся (то есть был модифицирован), то перед считыванием блока u в буфер содержимое последнего будет переписано во внешнюю память.
BUFFER	(u -- a)	Функции те же, что и у BLOCK, за исключением того, что сам блок из внешней памяти не считывается.

ОПЕРАТОРЫ ВЫВОДА

Слово **EMIT** берет из стека одно значение в коде ASCII, используя только младший байт, и выводит этот символ. Например, при текущей десятичной системе счисления вы получите такой результат:

```
65 EMIT Aok
66 EMIT Bok
```

Слово **TYPE** выводит всю строку символов при ее заданном начальном адресе и счетчике в следующей форме:

```
( a количество -- )
```

Вам уже встречалось слово **TYPE** в определениях, связанных с форматированием чисел, но тогда не нужно было заботиться ни об адресе, ни о счетчике, так как их значения обеспечивались словом #> автоматически.

Зададим слову **TYPE** адрес, по которому, как нам известно, находится строка символов. Напомним, что начальный адрес буфера входного текста задается словом **TIB** (см. гл. 9 о вариантах диалектов). Допустим, вы вводите команду:

```
TIB 11 TYPE
```

В результате будет выведено 11 символов из буфера входного текста, который содержит только что введенную команду:

```
TIB 11 TYPE<return>TIB 11 TYPEok
```

TYPE	(a количество --)	Происходит выдача заданного количества символов, начиная с заданного адреса, на текущее внешнее устройство

ВЫВОД ТЕКСТА С ДИСКА

Ранее уже отмечалось, что слово **BLOCK** копирует заданный блок в некоторый доступный буфер и оставляет адрес последнего в вершине стека. Приняв этот адрес за исходный, мы можем добраться посредством индексирования до каждого из 1024 байтов данного буфера и вывести любую строку. К примеру, для того чтобы вывести строку 0 блока 214, вы можете ввести

```
CR  214 BLOCK  64 TYPE<return>
( ЭТО БЛОК 214 )                                          ok
```

Для вывода строки 8 нужно добавить к исходному адресу 512 (8 × 64):

```
CR  214 BLOCK  512 + 64 TYPE
```

Прежде чем перейти к более интересному примеру, нам бы хотелось предложить вашему вниманию еще два слова, которые ассоциируются со словом **TYPE**.

-TRAILING	(a u1 -- a u2)	Удаление незначащих пробелов из строки с заданным адресом путем уменьшения значения счетчика от u1 (счечик исходных байтов) до u2 (счетчик байтов, полученных в результате вычеркивания пробелов),
>TYPE	(a # --)	То же самое, что и TYPE, за исключением того, что перед выдачей выводимая строка помещается в pad . Используется в мультипрограммных системах для вывода строк, находящихся в блоках на диске.

Слово **—TRAILING**, используемое непосредственно перед командой **TYPE**, изменяет значение счетчика таким образом, что незначащие пробелы не выводятся. Так, если вы вставите это слово в приведенный выше пример (с блоком 214), то получите:

```
CR 214 BLOCK  64 -TRAILING TYPE<return>
( ЭТО БЛОК 214 ) ок
```

Это слово удаляет незначащие пробелы в конце (32 в коде **ASCII**), но не препятствует другим не выводимым на печать символам (например, 0 в коде ASCII).

Слово **> TYPE** применяется только в мультипрограммных системах для вывода строк из буферов, находящихся на диске. Вме-

сто того чтобы непосредственно выдавать строку с заданного адреса, оно предварительно перекачивает строку целиком в рабочую область и затем выводит ее оттуда. Поскольку все пользователи разделяют одни и те же буферы, система не может гарантировать, что к тому времени, когда **TYPE** завершит вывод содержимого какого-то буфера, последний будет все еще хранить прежний блок. Однако вы можете быть уверены в том, что данный буфер содержит один и тот же блок во время перекачки этого буфера в рабочую область[1]. Так как каждой задаче отведена своя рабочая область, **>TYPE** может выводить из нее информацию без риска получить не те данные.

В приведенном ниже примере используется слово **TYPE**, но вы при необходимости можете подставить вместо него **>TYPE**. В конце раздела мы покажем вам полезный прием с применением генератора случайных чисел.

```
Block# 231

( Генератор бессмысленных сообщений )
: АБРЕД   ( -- а)   232 BLOCK ;
: БРЕД   ( строка# столбец# -- а)
   20 *   SWAP  64 * +   АБРЕД + ;
: .БРЕД   ( столбец# колонка# -- )   БРЕД  20 -TRAILING TYPE ;
: 1ПРИЛАГАТЕЛЬНОЕ    10 choose    0 .БРЕД ;
: 2ПРИЛАГАТЕЛЬНОЕ    10 choose    1 .БРЕД ;
: СУЩЕСТВИТЕЛЬНОЕ    10 choose    2 .БРЕД ;
: ФРАЗА   1ПРИЛАГАТЕЛЬНОЕ SPACE 2ПРИЛАГАТЕЛЬНОЕ SPACE СУЩЕСТВИТЕЛЬНОЕ ;
: СООБЩЕНИЕ
        CR  ." Применяя "  ФРАЗА  ." имея в виду "
        CR  ФРАЗА  ." представляется возможным даже несмотря на "
        CR  ФРАЗА  ." функционировать как "
        CR  ФРАЗА  ." при существующих ограничениях на "
        CR  ФРАЗА  ."  . " ;
СООБЩЕНИЕ

Block# 232

высокий             культурный          уровень
общий               производственный    интерес
автоматизированный  наукоемкий          комплекс
запланированный     валовой             объем
интегрированный     цифровой            коэффициент
квалифицированный   многоотраслевой     принцип
представительный    химический          генератор
автономный          аппаратный          интерфейс
цифровой            независимый         автомат
синхронизированный  функциональный      критерий
систематизированный коротковолновой     проект
```

[1] *Для специалистов.* В мультипрограммной системе задача передает управление центрального процессора следующей задаче только на время ввода-вывода или по специальной команде, которая преднамеренно не включена в определение слова, пересылающего строки.

После загрузки блока (в нашем примере блока 231) вы получите следующий текст, хотя некоторые слова при выполнении слова СООБЩЕНИЕ всякий раз будут меняться:

```
применяя высокий функциональный критерий имея в виду
цифровой производственный комплекс представляется возможным даже несмотря на
представительный независимый коэффициент функционировать как
автоматизированный наукоемкий уровень при существующих ограничениях на
квалифицированный функциональный автомат.
```

Как видите, определение слова СООБЩЕНИЕ состоит из ряда сочетаний ." текст", перемежаемых словом ФРАЗА. Если вы выполните слово ФРАЗА отдельно, то получите

```
ФРАЗА автоматизированный культурный объем ok
```

т. е. одно слово выбирается случайным образом из столбца 0 блока 232, другое — из столбца 1, а третье — из столбца 2.

Вы, конечно, заметили, что в определении слова ФРАЗА есть обращения к трем словам верхнего уровня: 1ПРИЛАГАТЕЛЬНОЕ, 2ПРИЛАГАТЕЛЬНОЕ и СУЩЕСТВИТЕЛЬНОЕ. Каждое из перечисленных слов в свою очередь обращается к слову .БРЕД, которому в качестве аргументов необходимо взять из стека номер строки (0—9) и номер столбца (0—2), определяющие выводимое слово или фразу. Случайное число, получаемое при выполнении выражения "10 **CHOOSE**", определяет номер строки. Каждая часть речи обеспечивает уникальный номер столбца.

Слово .БРЕД обращается к слову БРЕД для вычисления адреса бессмысленного сообщения, передавая максимальное значение счетчика, равное 20, слову **TYPE**. Но прежде слово **—TRAILING** сокращает значение 20 до фактического числа значащих символов в строке, удаляя тем самым незначащие пробелы в конце. Слово БРЕД вычисляет величину смещения в блоке путем умножения номера столбца на 20 (каждый столбец занимает 20 символов) и номера строки на 64 (каждая строка состоит из 64 символов), после чего складывает полученное смещение с адресом начала, которое доставляется словом АБРЕД. Вообще хороший стиль программирования предполагает разделение программ, вычисляющих адреса, и программ, эти адреса использующие (так как адреса зачастую оказываются необходимыми для других целей).

Адрес начала обеспечивает слово АБРЕД, которое просто вызывает **BLOCK**. Здесь мы снова выделили получение адреса в отдельное действие на тот случай, если местоположение базы данных с бессмысленными выражениями изменится. Например, может измениться номер блока. Такое разбиение не мешает нам даже поместить базу данных в словарь (путем переопределения слова АБРЕД).

```
CREATE АБРЕД  64 10 * ALLOT
```

Полезный прием. Генератор случайных чисел. Этот простой генератор случайных чисел может быть использован в играх. Для более же серьезных применений, например моделирования, существуют улучшенные варианты такого генератора.

```
( Генератор случайных чисел - верхний уровень )
VARIABLE  RND    HERE RND !
: RANDOM        ( -- )  RND @  31421 *  6927 +  DUP RND ! ;
: CHOOSE         ( u1 --u2 )   RANDOM UM*  SWAP DROP ;

( где CHOOSE оставляет на стеке случайное число в диапазоне
    от 0 до u1-1 )
```

Для получения некоторого случайного числа в диапазоне от 0 до 10 (само число 10 сюда не входит) просто наберите на клавиатуре

```
10 CHOOSE
```

и в вершине стека останется случайное число.

ОПЕРАЦИИ НАД СТРОКАМИ В ОПЕРАТИВНОЙ ПАМЯТИ

Команды для пересылки строк литер или массивов данных весьма просты. Каждой команде требуются три аргумента: исходный адрес, конечный адрес и значение счетчика.

MOVE	(a1 a2 u --)	Поячеечное копирование участка памяти длиной u байтов, начинающегося с адреса a1, в участок памяти, начинающийся с a2. Пересылка идет с a1 в сторону увеличения адресов.
CMOVE	(a1 a2 u --)	Побайтное копирование участка памяти длиной u байтов, начинающегося с a1, в участок памяти, начинающийся с a2. Пересылка идет с a1 в сторону увеличения адресов.
CMOVE>	(a1 a2 u --)	Копирование участка памяти длиной u байтов, начинающегося с адреса a1, в участок памяти, начинающийся с адреса a2. Копирование начинается с КОНЦА строки и продвигается в сторону уменьшения адресов.

Заметим, что для этих команд существуют определенные соглашения (рассмотренные ранее):

- если среди аргументов находятся источник и адресат, как это имело место в случае с **COPY**, то источник предшествует адресату;
- если среди аргументов находятся адрес и счетчик, как это имело место в случае с **TYPE**, то адрес предшествует счетчику.

Поэтому для перечисленных трех слов аргументы следуют в таком порядке: (источник адресат счетчик --). Для того чтобы переслать содержимое всего буфера в рабочую область, вы могли бы, к примеру, написать:

```
210 BLOCK  PAD  1024 CMOVE
```

хотя на машинах с поячеечной адресацией памяти намного быстрее выполнялась бы последовательная пересылка ячеек (ячейка за ячейкой):

```
210 BLOCK  PAD  1024 MOVE
```

Слово **CMOVE>** позволяет пересылать некоторую строку в область, которая расположена в памяти с бóльшими адресами, но пересекается с исходной областью[1].

Для заполнения некоторого массива пробелами вы можете воспользоваться уже знакомым вам словом **BLANK**[2]. Например, внесите пробелы в 1024 байта рабочей области:

```
PAD 1024 BLANK
```

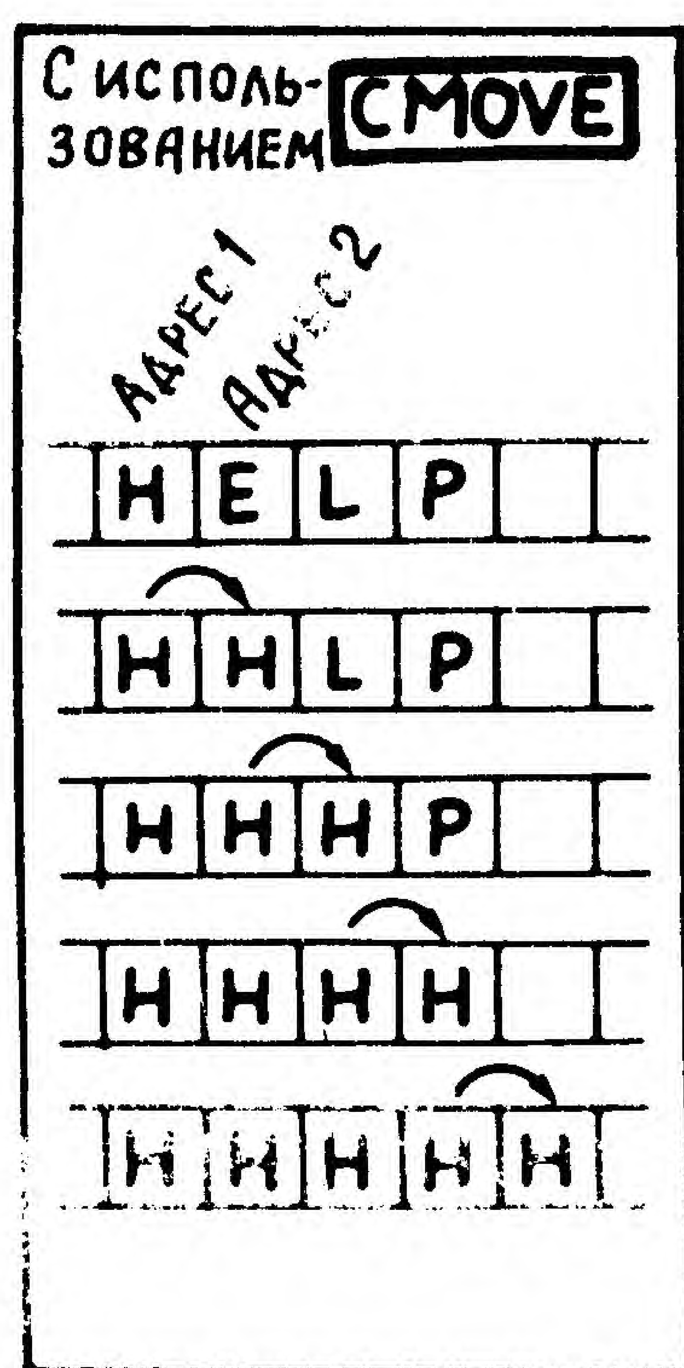
Это равносильно выражению:

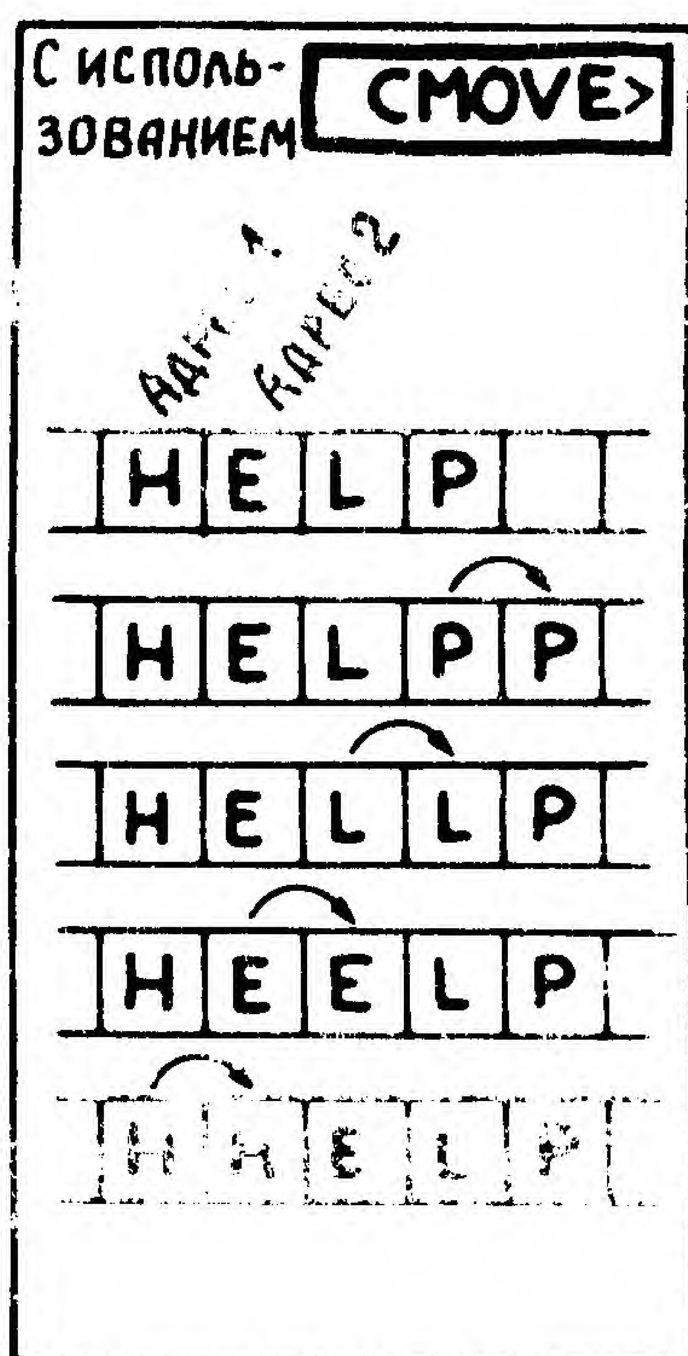
```
PAD  1024 BL FILL
```

В большинстве систем слово **BL** определено как константа со значением 32.

---

[1] *Для начинающих.* Предположим, вы хотите передвинуть некоторую строку в памяти на один байт вправо (например, когда текстовый редактор включает в текст некоторый символ). Применив команду **CMOVE**, вы скопировали бы первую букву строки во второй байт, но при этом вторая литера строки оказалась бы «испорченной». В результате получилась бы строка, составленная из одного и того же символа. Чтобы сохранить исходную строку, в данной ситуации нужно воспользоваться словом **CMOVE>**.





[2] *Для пользователей систем фиг-Форта.* В вашей системе это слово **BLANKS** (ПРОБЕЛЫ).

```
BLANK          ( a u -- )  Заполнение участка памяти длиной u байт
                           символом пробела в коде ASCII.
```

Остроумный, но не согласующийся со стандартом прием: для заполнения участка памяти многобайтным шаблоном примените слово **CMOVE**. При выполнении следующего выражения 20-байтный массив будет заполнен 10-ю копиями 16-разрядного адреса слова НОП:

```
CREATE ТАБЛИЦА  ' НОП ,  18 ALLOT \  10 ячеек
ТАБЛИЦА  DUP 2+ 18 CMOVE  \  инициализация выделенной памяти
\  адресом НОП
```

ВВОД С КЛАВИАТУРЫ

Слово **KEY** (КЛАВИША) ожидает, пока вы нажмете какую-либо клавишу на панели терминала и оставляет в вершине стека в младшем по порядку байте эквивалент символа, соответствующего этой клавише, в коде ASCII.

Наберите на клавиатуре:

```
KEY<return>
```

Курсор продвинется на одну позицию, но «ok» на терминале *не высветится*: система ждет ввода вашего символа. Нажмите, к примеру, клавишу А и Форт-система ответит вам: «ok». Теперь в вершине стека находится значение литеры А в коде ASCII, поэтому введите

```
.<return> 65 ok
```

Это дает вам возможность определить значение символа в коде ASCII, не заглядывая в таблицу.

Вы можете также включить **KEY** в состав определения. При встрече слова **KEY** выполнение данного определения приостановится до тех пор, пока не будет введен некоторый символ. Например, следующее определение выводит на печать по порядку заданное число блоков, начиная с текущего, но, прежде чем приступить к распечатке очередного блока, ожидает нажатия любой клавиши:

```
: БЛОКИ  ( # -- )
   SCR @ +  SCR @ DO  I LIST  KEY DROP  LOOP ;
```

В этом случае вы снимаете со стека посредством **DROP** значение, оставленное словом **KEY**, так как оно вам не нужно. Позднее мы продемонстрируем использование слова **KEY** на примере программы ввода.

В ряде систем имеется нестандартное слово с именем **KEY?** (в более ранних системах **?TERMINAL**), которое помещает в вершину стека значение истины при нажатии на одну из клавиш, не останавливая вычисления и не ожидая ввода самого символа.

Допустим, вы выполняете бесконечный цикл на прибавление единицы:

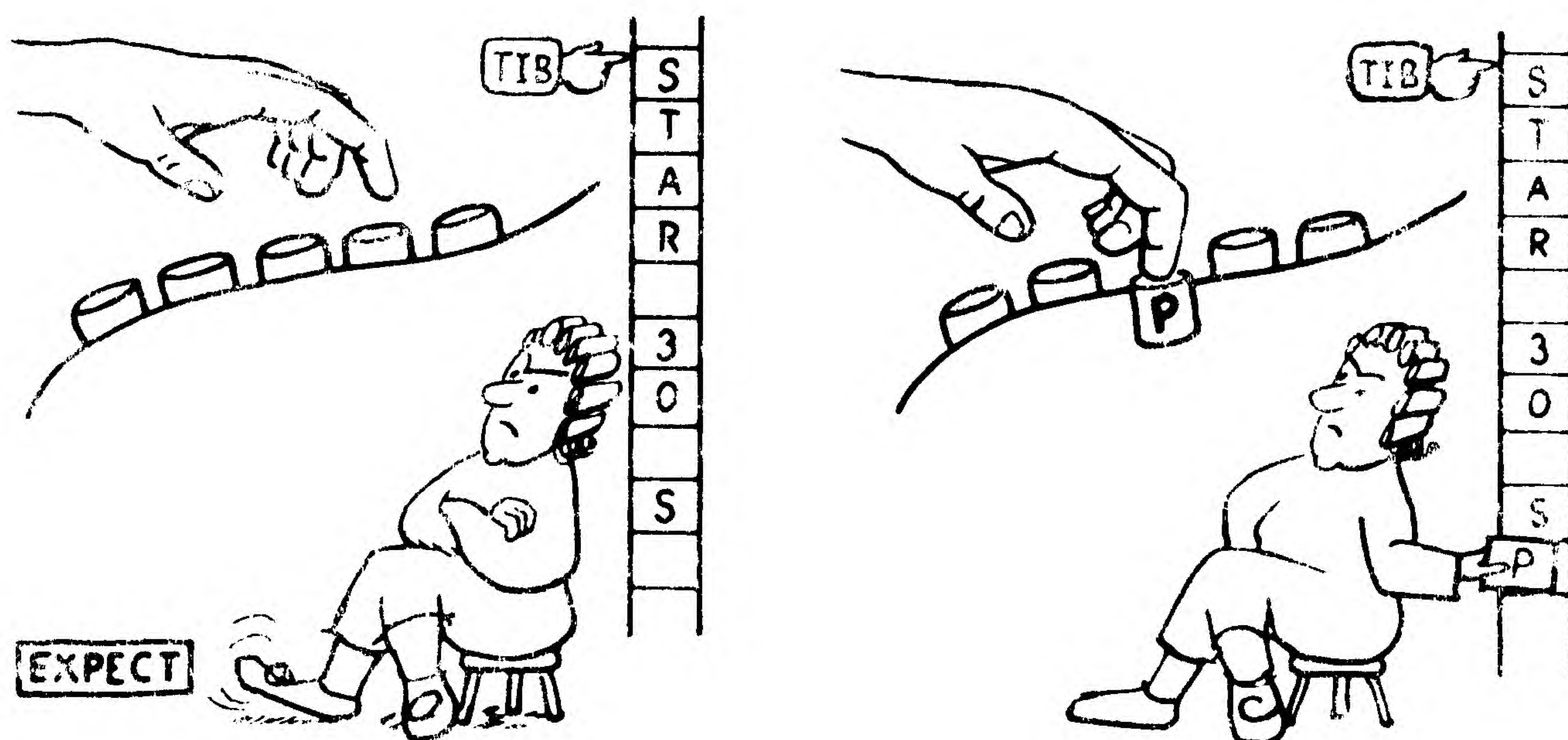
```
: БЕСКОНЕЧНЫЙ    Ø BEGIN  DUP . 1+  FALSE UNTIL  DROP ;
```

Можно организовать выход из такого цикла, заменив слово **FALSE** на **KEY?**:

```
: СКАЖИ-КОГДА   Ø BEGIN   DUP . 1+  KEY? UNTIL  DROP
   KEY DROP ;
```

Слово **KEY?** не считывает значение символа, соответствующего нажатой клавише, а лишь сигнализирует о том, что одна из клавиш была нажата. Чтобы считать ее значение, вы должны обратиться к слову **KEY**. В этом месте вы можете либо определить значение нажатой клавиши (при необходимости), либо просто выполнить выражение KEY DROP. (Во многих системах, где есть этап предварительного чтения в буфер, невозможность применять связку **KEY KEY?**, по существу, сводит на нет описываемый универсальный прием и вынуждает прибегать к системно-зависимым «ухищрениям»).

Если слово **KEY** ожидает ввода одного символа, то **EXPECT** (ОЖИДАТЬ) ожидает ввода с клавиатуры целой строки. На самом деле это адекватно использованию **KEY** в цикле. Цикл заканчивается по достижении заданного числа нажатий клавиш (обычно 80) или при нажатии клавиши возврата каретки. Кроме того, слово **EXPECT** способно распознать значение клавиши «Забой» и возвратить назад как курсор, так и внутренний указатель слова. С помощью **EXPECT** Форт-система ожидает ввода вашей команды.



EXPECT выбирает из стека два аргумента: адрес, по которому нужно запомнить вводимый текст, и максимальное значение счетчика. Например, выражение

```
TIB 80 EXPECT
```

ожидает ввода до 80 символов или нажатия клавиши RETURN и после завершения набора помещает введенный текст в буфер входного текста. Приведенное выражение содержится в определении слова **QUERY** (ЗАПРОС), используемое, как было показано выше, словом **QUIT**.

С помощью **EXPECT** вы можете сделать запрос на ввод из определения[1]. Ниже дается слово, которое при своем выполнении запрашивает имя пользователя, а затем выводит введенное имя вместе с приветствием:

```
CREATE ИМЯ-ПОЛЬЗОВАТЕЛЯ  40 ALLOT
: .ПОЛЬЗОВАТЕЛЬ   ИМЯ-ПОЛЬЗОВАТЕЛЯ 40 -TRAILING TYPE ;
: ПОЛУЧЕНИЕ-ИМЕНИ    ИМЯ-ПОЛЬЗОВАТЕЛЯ 40 BLANK  ИМЯ-ПОЛЬЗОВАТЕЛЯ
    40 EXPECT ;
: ВСТРЕЧА   CR  ." Пожалуйста, введите свое имя: " ПОЛУЧЕНИЕ-ИМЕНИ
    CR  ." Привет, " .ПОЛЬЗОВАТЕЛЬ  ." , Я говорю на Форте.  " ;
```

В результате вы получаете[2]:

```
ВСТРЕЧА_
Пожалуйста, введите свое имя: ВАСЯ
Привет, ВАСЯ, Я говорю на Форте.
```

---

[1] *Для специалистов.* Вы можете использовать **EXPECT** для снятия данных с последовательной шины, например, некоторого измерительного устройства. Так как у вас задействованы адрес и счетчик, такие данные могут быть считаны непосредственно в массив. Если вы являетесь единственным пользователем системы, то перед записью на диск можете считать данные в буфер. В случае же мультизадачной системы вы должны применить **TIB** и уже потом пересылать данные в указанный буфер, поскольку «вашим» буфером может воспользоваться другая задача.

[2] *Для пользователей систем, созданных до введения Стандарта-83.* Слово **EXPECT** в таких системах требует наличия нуля в конце вводимого текста. Поэтому при выполнении приведенного выше примера на вашей системе между именем и запятой может появиться пробел. **-TRAILING** здесь воспринимает нуль как невыводимый на печать символ и при его выводе печатается пробел. Во избежание этого нужно ввести текст посредством **EXPECT** в рабочую область (**PAD**), после чего скопировать его, используя **SPAN**, в слово ИМЯ-ПОЛЬЗОВАТЕЛЯ с требуемым числом символов:

```
: ПОЛУЧЕНИЕ-ИМЕНИ  ИМЯ-ПОЛЬЗОВАТЕЛЯ 40 BLANK  PAD 40 EXPECT
   PAD ИМЯ-ПОЛЬЗОВАТЕЛЯ   SPAN @ CMOVE ;
```

SPAN — пользовательская переменная, в которой содержится фактическое число символов, полученных словом **EXPECT**.

```
KEY       ( -- c )      Занесение на стек значения в коде ASCII очеред-
                        ного доступного символа на текущем устройстве
                        ввода.

EXPECT    ( a ц -- )    Ожидание ц символов (или нажатий клавиши  RETURN )
                        с клавиатуры и запоминание их в участок памяти,
                        начинающийся с адреса а и продолжающийся в сторону
                        увеличения адресов. На нажатие клавиши  ЗАБОЙ
                        осуществляется возврат курсора.

SPAN      ( -- a )      Содержится количество символов, полученных словом
                        EXPECT
```

ВВОД ИЗ ВХОДНОГО ПОТОКА

Мы рассмотрели выше, каким образом ожидается ввод с клавиатуры. Однако Форт-система позволяет осуществлять ввод и из входного потока. Напоминаем, что входной поток — это последовательность символов, предназначенных для обработки текстовым интерпретатором. Символы могут быть расположены в буфере входного текста (в режиме интерпретации) или в блоке (в режиме загрузки).

Предположим, что пользователь хотел бы иметь возможность задавать свое имя, используя слово Я, например:

```
Я ВАСЯ<return>
```

Пользователь должен набрать фрагмент Я ВАСЯ на одной строке, а затем нажать клавишу RETURN. Нам нужно, чтобы слово Я помещало имя пользователя в массив ИМЯ-ПОЛЬЗОВАТЕЛЯ. Но фрагмент ВАСЯ находится впереди по входному потоку и слово Я, следовательно, не может «ожидать» его с помощью **EXPECT**, так как он уже введен. Вместо этого необходимо найти средство для *чтения* опережающего слова.

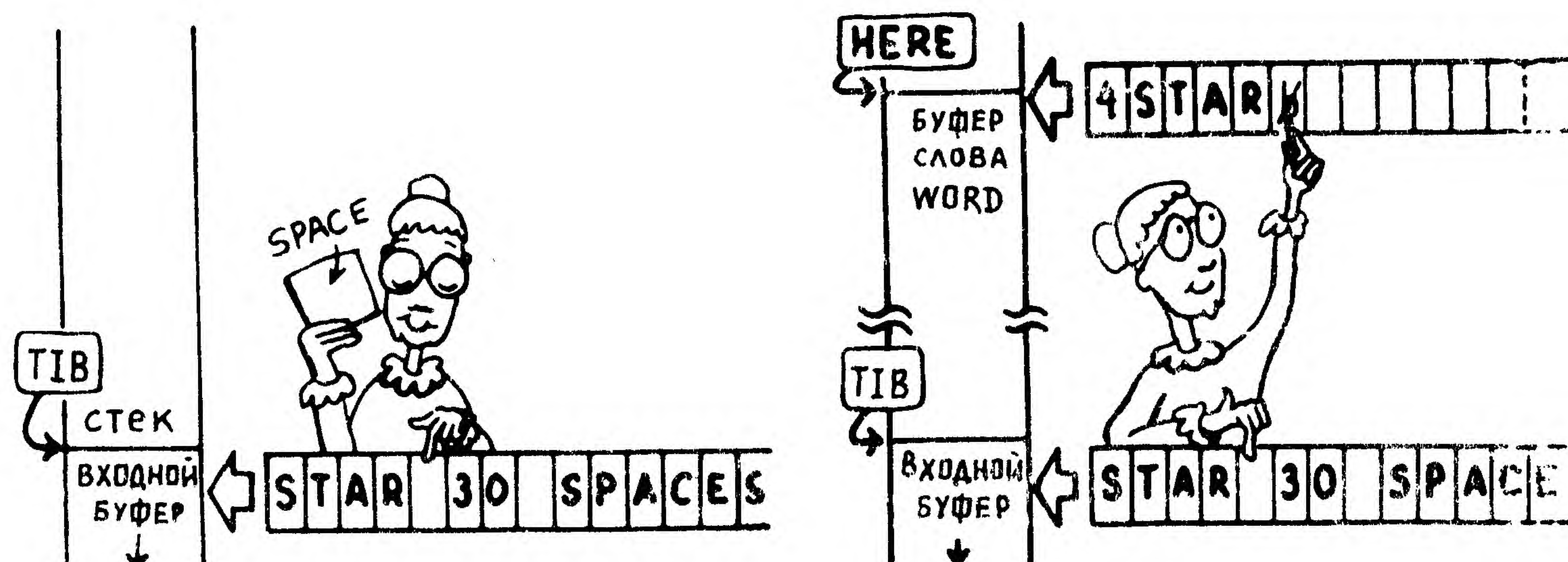
Таким средством является слово **WORD** (СЛОВО). Оно сканирует входной поток в поисках фрагмента текста, ограниченного символом, код ASCII которого хранится в вершине стека. Например, выражение

```
BL WORD
```

будет просматривать входной поток в поисках фрагмента текста, ограниченного пробелами. Найденную подстроку **WORD** поместит в свой собственный временный буфер вместе со счетчиком симво-

лов в первом байте буфера. В Форте строка символов, предваряемая одним байтом, в котором содержится число символов данной строки, называется *строкой со счетчиком*. Затем **WORD** вносит в вершину стека адрес своего временного буфера[1].

Слово **WORD** — важный элемент текстового интерпретатора Форта, в котором выражение BL WORD применяется для сканирования входного потока в поисках слов и чисел.



Пересылая фрагмент текста, **WORD** дополняет его в конце пробелом, но этот пробел не учитывается в счетчике. Вы можете создать определение Я следующим образом (используя массив, созданный ранее для слова ВСТРЕЧА):

```
: Я  (  имя-пользователя    ( -- )
   ИМЯ-ПОЛЬЗОВАТЕЛЯ 40 BLANK  BL WORD COUNT ИМЯ-ПОЛЬЗОВАТЕЛЯ
   SWAP CMOVE ;
```

Чтобы видеть вводимое имя, можно воспользоваться ранее определенным словом .ПОЛЬЗОВАТЕЛЬ.

Каковы функции слова **COUNT**? Это слово разделяет адрес строки со счетчиком на адрес и счетчик. Полученный адрес указывает начало текста, а не байт со счетчиком. Например, при заданном в вершине стека адресе строки со счетчиком **ПРИВЕТ**

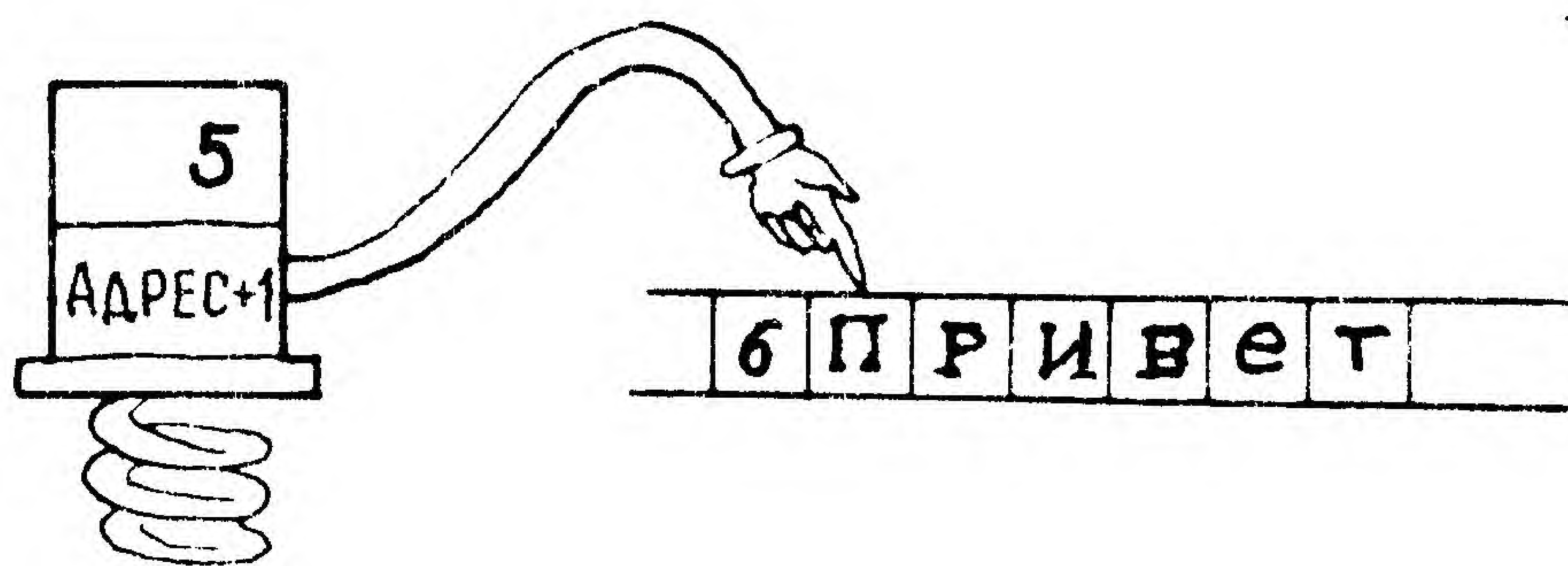


---

[1] *Для пользователей систем фиг-Форта.* Ваша версия слова **WORD** ничего в вершине стека не оставляет. Для обеспечения совместимости с текстом рассматриваемого примера переопределите его:

```
: WORD  ( -- a)  WORD HERE ;
```

слово **COUNT** заносит в стек счетчик, увеличивает адрес:



и в вершине стека остаются адрес строки и значение счетчика, которые могут служить аргументами для слов **TYPE**, **CMOVE** и т. д.

В нашем определении Я слово **WORD** оставляет в вершине стека адрес строки со счетчиком, а слово **COUNT** разделяет этот адрес на адрес и счетчик. Слово АДРЕС-ПОЛЬЗОВАТЕЛЯ обеспечивает адрес назначения. Слово **SWAP** расставляет аргументы в требуемом порядке (источник — получатель — счетчик) для **CMOVE**.

Обратите внимание на необычную стековую нотацию слова Я. По существующему соглашению описание, предшествующее стековому комментарию, указывает фрагмент, поиск которого во входном потоке осуществляется в данном определении. Может применяться также символ « »:

```
: Я  \  имя-пользователя   ( -- )
  ИМЯ-ПОЛЬЗОВАТЕЛЯ 40 BLANK  BL WORD  COUNT  ИМЯ-ПОЛЬЗОВАТЕЛЯ
  SWAP CMOVE ;
```

Отметим две особенности слова **WORD**. Первая особенность связана с тем, что, поскольку это слово используется текстовым интерпретатором Форта, найденный им фрагмент будет затерт при чтении следующего фрагмента из входного потока. Введите выражение

```
BL WORD HI COUNT TYPE
```

Выражение BL WORD прочитает фрагмент HI и поместит его во временный буфер, но во время интерпретации слова **COUNT** фрагмент HI будет затерт фрагментом COUNT и в первом байте окажется значение счетчика 5, а не 2. При выполнении слова **COUNT** в вершину стека заносится значение счетчика 5. Наконец, при интерпретации слова **TYPE** фрагмент TYPE затрет предыдущий и выведется на экран (включая пробел пятым символом).

Поскольку **WORD** обычно находится внутри определения, фрагмент, полученный в результате выполнения этого слова, нужно переслать из буфера последнего в более надежное хранилище до считывания из входного потока очередного слова.

Другая особенность слова **WORD** состоит в том, что оно не воспринимает начальные вхождения символа-ограничителя. Если

в начале некоторого фрагмента набраны пробелы или если слова разделены пробелами, то выражение BL WORD осуществляет поиск до первого значащего символа и считывает фрагмент до первого пробела. Помещаются во временный буфер и учитываются в счетчике только значащие символы. Указанная особенность может вызывать затруднения при работе со словом **WORD**. Например, в гл. 3 было введено слово **.(**, обеспечивающее непосредственный вывод на экран очередного фрагмента из входного потока до правой круглой скобки. Это слово может быть определено следующим образом:

```
: .(    \     текст)    ( -- )
   ASCII ) WORD COUNT TYPE ;
```

Комментарий означает, что в данном определении фрагмент будет считываться до правой круглой скобки «)». Но в таком определении не предусмотрена ситуация с пустой строкой (без символов):

```
.( )  CR CR
```

Наше определение не воспримет правую круглую скобку, поскольку она является первым просматриваемым символом, а посчитает фрагмент CR CR за строку, которая должна быть выведена на экран. Для подобных ситуаций в ряде Форт-систем имеется слово **PARSE** (РАЗБОР), функционирующее аналогично слову **WORD**, но воспринимающее начальные вхождения символа-ограничителя. Помните, что **PARSE** оставляет в вершине стека адрес строки и значение счетчика, а не адрес строки со счетчиком, как это делает **WORD.**

Ниже приводится слово, которым вы можете воспользоваться:

```
: TEXT  ( c)  PAD 80 BLANK  WORD COUNT  PAD SWAP CMOVE> ;
```

Подобно **WORD**, слово **TEXT** выбирает из стека символ-ограничитель и сканирует входной поток до тех пор, пока из него не будет считан фрагмент, ограниченный этим символом. Затем фрагмент помещается в рабочую область (PAD). Отличительной чертой **TEXT** является то, что рабочая область перед занесением строки заполняется пробелами, что облегчает выполнение слов **TYPE** и **—TRALLING**.

WORD	(c -- a)	Чтение слова, ограниченного заданным символом, из входного потока. Полученный фрагмент оформляется в виде строки со счетчиком и ее адрес помещается в стек.

```
COUNT   ( a -- a+1 #)  Преобразование адреса строки со счетчиком
                       (длина которой находится в первом строки) в
                       формат, соответствующий использованию словом
                       TYPE, а именно: в стек заносится адрес начала
                       текста строки и ее длина.
```

ПРИМЕНЕНИЕ СЛОВА WORD

Помимо текстового интерпретатора многие слова Форта используют **WORD**. Так, слово **CREATE** выбирает из входного потока имя создаваемого слова, а **FORGET** — имя слова, которое должно быть забыто.

Вернемся к приведенному выше примеру с генератором бессмысленных сообщений. Хотелось бы иметь при таком генераторе удобные средства введения в базу данных бессмысленных фраз (текстовый редактор Форта не показывает нам, где начинается 20-й или 40-й столбец). Допустим, нам нужно определить для введения в базу данных очередного слова из входного потока слово «добавить»:

```
начало
добавить высокий
добавить культурный
добавить уровень
добавить общий
и т.д.
```

Это можно сделать следующим образом:

```
\ загрузчик в базу данных бессмысленных сообщений
VARIABLE РЯД
VARIABLE СТОЛБЕЦ
: начало   0 РЯД !  0 СТОЛБЕЦ ! ;
: +РЯД   1 РЯД +! ;
: +СТОЛБЕЦ   СТОЛБЕЦ @  1+  3 /MOD  РЯД +!  СТОЛБЕЦ ! ;
: добавить   \   бессмысленная фраза   ( -- )
   1 WORD COUNT  РЯД @ СТОЛБЕЦ @ БРЕД  DUP 20 BLANK
   SWAP CMOVE  UPDATE  +СТОЛБЕЦ ;
```

Обратите внимание на то, как приведенные выше программы вычисляют соответствующие строку и столбец, а также на удачное выделение слова БРЕД, которое оставляет в вершине стека адрес в блоке пересечения заданных строки и столбца.

Кроме того, необходимо отметить, что выражение 1 WORD удачнее выражения BL WORD, поскольку некоторые фразы состоят из двух слов, разделенных пробелами, а мы не хотим считывать только первое слово. Наша цель — считать все, что пользователь ввел до конца строки. В коде ASCII символ 1 является управляющим. Обычно он не может быть введен с клавиатуры и,

значит, не может появиться среди символов входного буфера. Поэтому выражение 1 WORD применяется для чтения содержимого входного буфера до того момента, пока пользователь не нажмет клавишу RETURN.

Со словом **WORD** могут сочетаться и другие ограничители, например кавычки и круглые скобки. Слово Форта ." использует выражение

```
ASCII " WORD
```

для чтения из входного потока выводимой строки. Слово (использует выражение

```
ASCII ) WORD
```

для выборки из входного потока фрагмента, который должен быть пропущен. Считывая из входного потока запятые или иные разделители, можно даже обрабатывать несколько фрагментов из одной строки, но из разных полей.

Уже знакомое вам слово **TEXT** упрощает определение слова «добавить»:

```
: добавить    \   бессмысленную фразу   ( -- )
   1  TEXT  PAD  РЯД @ СТОЛБЕЦ @ БРЕД  20 CMOVE UPDATE
   +СТОЛБЕЦ ;
```

УКАЗАТЕЛИ ВХОДНОГО ПОТОКА, ИСПОЛЬЗУЕМЫЕ СЛОВОМ WORD

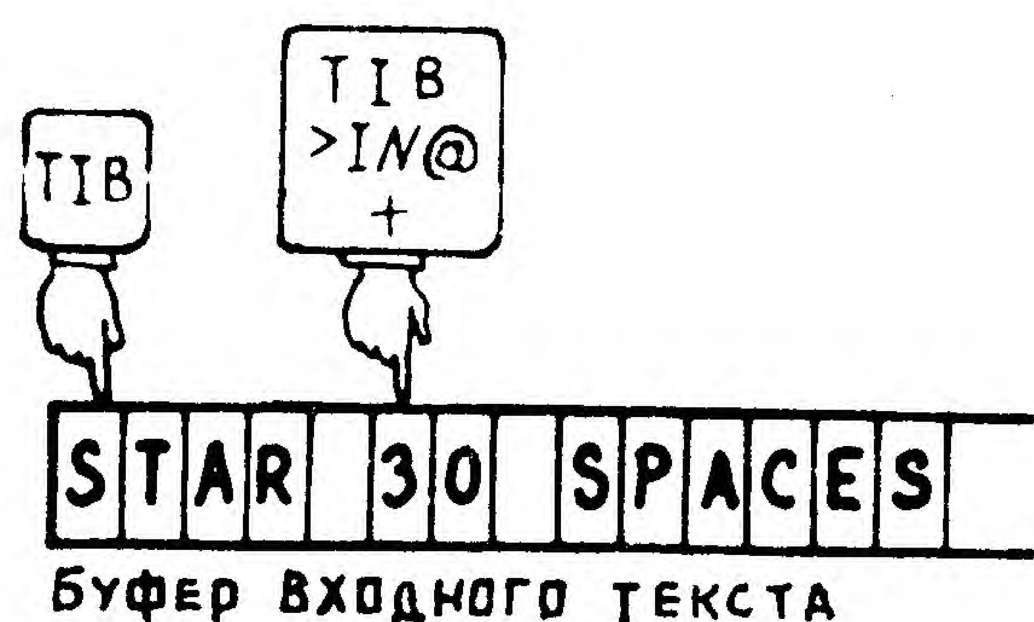
Откуда и что читать, слово **WORD** «понимает» по двум указателям. Первый из них **>IN**[1] — *относительный указатель*. Он показывает, на сколько байтов относительно начала буфера входного потока уже продвинулся интерпретатор. Допустим, вы ввели текст

```
STAR 30 SPACES
```

и нажали клавишу RETURN. Вначале переменная **>IN** устанавливается в нуль. После того как слово **WORD** прочитало строку STAR из входного потока, значение **>IN** становится равным 5.

[1] *Для пользователей систем фиг-Форта.* В вашей системе это слово IN. Чтобы обеспечить соответствие нашему указателю, переопределите его:

```
: >IN  ( -- a)   IN ;
```



В своих программах вы можете изменять содержимое **>IN** с тем, чтобы установить необходимый вам порядок интерпретации слов.

Второй указатель — **BLK**. Вспомните, что входной поток представляет собой последовательность символов, которые могут быть либо в буфере входного текста, либо в загружаемом блоке. Слово **BLK** указывает, где именно они находятся. Это слово играет роль и флага, и указателя. Если **BLK** содержит нуль, то **WORD** сканирует буфер входного текста, а если ненулевое значение, то **WORD** сканирует блок, номер которого находится в **BLK** (поэтому вы не можете загружать блок 0).

Ниже показан адрес интерпретируемого в данный момент текста:

СОДЕРЖИМОЕ BLK	ТЕКУЩИЙ АДРЕС, ИСПОЛЬЗУЕМЫЙ СЛОВОМ WORD
Нуль	TIB >IN + (>IN байт относительно начала буфера входного текста)
Не нуль	BLK @ BLOCK >IN @ + (>IN байт относительно начала блочного буфера)

На Форте адрес сканирования для слова **WORD** вычисляется следующим образом:

```
... BLK @ ?DUP IF  BLOCK  ELSE  TIB THEN  >IN @ + ...
```

Заметьте, что **WORD** обращается к слову **BLOCK**, поэтому если текст интерпретируется из блока, то его содержимое обязательно уже находится в некотором буфере.

>IN	(— a)	Пользовательская переменная, содержащая смещение обрабатываемого символа относительно начала входного потока.
BLK	(— a)	Пользовательская переменная, содержащая номер блока во внешней памяти, интерпретируемого в качестве входного потока. Если в BLK содержится нуль, то входной поток поступает из буфера входного текста.

ПРЕОБРАЗОВАНИЕ ВВОДИМЫХ ЧИСЕЛ

Как было показано в гл. 7, с помощью слов < # и # > можно преобразовать число, находящееся в стеке, в строку символов в коде ASCII. Слово **CONVERT** (ПРЕОБРАЗОВАТЬ) выполняет обратную функцию: оно переводит строку символов ASCII, представляющих некоторое число, в двоичную форму и заносит его в стек.

```
CONVERT  ( ud1 a1 --  Начиная с адреса a1+1 (байт, содержащий длину,
           ud2 a2)    пропускается), CONVERT преобразует строку в
                      двоичное значение, которое зависит от текущей
                      системы счисления (значения BASE). Полученное
                      значение накапливается  в  ud1, и
                      остается на стеке как ud2. Процесс продолжает-
                      ся до тех пор, пока не встретится символ, кото-
                      рый не может быть истолкован как цифра в теку-
                      щей системе счисления. Адрес этого символа за-
                      носится на стек как a2.
```

Приведем несложный пример:

```
: ПЛЮС   \   n2   ( n1 -- сумма )
   0 0  BL WORD  CONVERT 2DROP  + ;
```

Это определение можно использовать следующим образом:

```
12 ПЛЮС 23 . 35 ok
```

Слово ПЛЮС является лучшим доказательством (для скептиков) того, что в Форте можно применять инфиксную запись выражений. Это слово начинает выполняться при одном занесенном в стек в двоичной форме аргументе 12. В первую очередь, занося в стек нуль двойной длины, мы очищаем место для накопления, после чего слово **WORD** сканирует входной поток для чтения второго аргумента n2 и оставляет адрес прочитанного фрагмента в вершине стека. Слово **CONVERT** преобразует строку по заданному адресу (пропуская первый байт со счетчиком) в число двойной длины и заносит его в стек на место нуля. **2DROP** удаляет из стека последний адрес, помещенный в него словом **CONVERT** вместе со старшей ячейкой преобразованного числа, превращая последнее тем самым в число одинарной длины. Наконец, + складывает два верхних числа в стеке.

Если бы слово **CONVERT** функционировало с такими аргументами:

```
( a -- ud)
```

то его можно было бы применять повторно, выполняя преобразования строки, содержащей различные нецифровые символы. Например, строку

```
6/20/85
```

можно преобразовать в три числа одинарной длины, трижды подряд обращаясь к **CONVERT**. Адрес, оставленный в вершине стека при первом применении **CONVERT** передавался бы аргументом второму **CONVERT** и т. д.

В большинстве систем имеется слово **NUMBER** (число), которое выполняет те же функции, но зачастую проще в употреблении. В Стандарте-83 (слова несогласованного набора) это слово определено следующим образом:

```
NUMBER       ( a -- d)    Преобразование текста, начинающегося с ад-
                          реса a+1, в двоичное значение с учетом те-
                          кущей системы счисления (значения BASE).
                          Строка может предваряться знаком минус,что
                          делает полученное значение отрицательным.
```

Поэтому слово ПЛЮС лучше определить так:

```
: ПЛЮС  \  n2   ( n1 -- сумма)
   BL WORD  NUMBER DROP  + ;
```

Слово **NUMBER** используется и самой ФОРТ-системой. Это «обработчик чисел», к которому обращается текстовый интерпретатор, если искомое слово не найдено в словаре. **NUMBER** пытается преобразовать полученный фрагмент в число и в случае удачи заносит его значение в стек, при неудаче же осуществляется **ABORT**.

В каждой Форт-системе процесс преобразования чисел происходит по-своему, так как способов их введения существует очень много. Ниже будет показано одно из возможных определений слова **NUMBER**, которое воспринимает символы

```
:  ,  -  .  /
```

как правильные пунктуационные знаки, указывающие, что данный фрагмент нужно считать числом двойной длины. Если внутри какого-либо числа появился один из перечисленных символов, то в переменную **DPL** (положение десятичной точки) заносится количество цифр в числе справа от точки. Например, при вводе фрагмента 200.2 **DPL** содержит единицу. Если в числе *нет знаков пунктуации*, то значение **DPL** окажется равным —1.

Определение слова NUMBER

```
: NUMBER? ( адр -- d t=успешное-завершение)

DUP 1+ C@                      Получение первой цифры

ASCII - =                      Это знак минус?

DUP >R                         Запоминание флага в стеке возвратов

                               Если первым символом является "-", то
                               к адр добавляется 1, чтобы тот указы-
                               вал на первую цифру ( вычитание -1 рав-
                               носильно прибавлению 1)

-1 DPL !                       Отметка того, что знаков пунктуации
                               пока нет.

0 0 ROT                        В качестве первоначального накапливаемого
                               значения берется 0 двойной длины.

BEGIN CONVERT                  Преобразование до первого символа, не
                               являющегося цифрой.
DUP C@ DUP ASCII :=            это двоеточие ?
SWAP ASCII , ASCII / 1+        Это запятая, дефис, точка или слэш?
   WITHIN OR

WHILE  0 DPL !  REPEATE        Если да, то переустановить DPL и продолжать.

-ROT R> IF DNEGATE THEN        Перемещение d в вершину. Если значение
                               в стеке возвратов указывает минус, то
                               число делается отрицательным.

ROT C@ BL = ;                  Является ли последний, непреобразованный,
                               символ пробелом, как это и должно быть?

: NUMBER  ( адр -- d)

   NUMBER?  NOT ABORT" ?" ;    Если преобразование завершилось неудачей,
                               то аварийный выход посредством ABORT.
```

В приведенном определении учитывается, что **CONVERT** вычитает из **DPL** по единице при обработке каждой цифры до тех пор, пока значение переменной не станет равным —1. Кроме того, в определении используется слово **WITHIN**, аналогичное слову ВНУТРИ (см. упражнение к гл. 4).

В качестве «истины» принято арифметическое значение — 1, как это определено Стандартом-83. Для более ранних систем, где значением истины является единица, в строке 4 нужно заменить «—» на «+».

В Форте число, вводимое без знаков пунктуации, заносится в стек как число одинарной длины. При рассмотренном здесь определении слова **NUMBER** текстовый интерпретатор должен обращаться к нему примерно так:

```
... NUMBER  DPL @ -1 = IF DROP THEN ...
```

ПОСТРОЕНИЕ ПРОГРАММЫ ВВОДА ЧИСЕЛ С ПОМОЩЬЮ СЛОВА KEY

В данном разделе мы покажем вам, как слово **KEY** может быть использовано при разработке специализированного интерпретатора ввода с клавиатуры. Допустим, вы хотите создать слово, подобное **EXPECT**, которое ожидало бы ввода с клавиатуры, но воспринимало бы только числовые символы. Остальные символы не должны восприниматься. Исключение составляют сигналы, посылаемые при нажатии клавиш ЗАБОЙ (в обычном значении) и RETURN (конец ввода). В отличие от **EXPECT** ваше слово не будет завершать свое выполнение при вводе заданного числа цифр, так как пользователь может, нажав клавишу ЗАБОЙ, исправить последнюю цифру до нажатия клавиш RETURN. Назовем это слово EXPECT # и зададим ему следующий порядок аргументов:

```
( а макс-длина -- факт-длина)
```

где а — поле для размещения строки, а макс-длина — предельное число вводимых цифр. Фактическая длина может понадобиться в том случае, если вы захотите проверить, ввел ли пользователь хотя бы одну цифру.

Вам придется осуществлять такие действия, как возврат курсора на одну позицию, что программируется в разных системах по-разному. Принято вычленять подобные фрагменты в слова Форта и определять их отдельно от остальной части программы. Это делает программу более *мобильной*, поскольку при смене системы достаточно будет заменить лишь соответствующие слова. Такой прием называется *локализацией* или *упрятыванием* информации, не существенной для пользователя. Для большинства компьютеров мы можем написать определение:

```
: ЗАБОЙ   8 EMIT ;
```

но в некоторых системах, где применяются устройства вывода с распределенной памятью, могут потребоваться другие определения. Кроме того, символы, получаемые словом **KEY** при нажатии клавиш ЗАБОЙ и RETURN, в различных системах различны. Упрятывание информации происходит следующим образом:

```
: ЗБ?  ( c -- t=клавиша-забоя) 8 = ;  \ в некот.сист. 12
: ВК?  ( c -- t=клавиша-RETURN) 13 = ;  \ в некот.сист. 0
```

Основной конструкцией слова EXPECT # является цикл **BEGIN WHILE REPEAT**. Здесь **WHILE** выполняет проверку на нажатие клавиши RETURN. Цикл начинается со слова **KEY**. Если введена

правильная цифра, вы обрабатываете ее с помощью слова ПОЛУЧЕНИЕ (посылаете в буфер и увеличиваете соответствующий указатель). В случае «ЗАБОЯ» вы уничтожаете последний символ посредством слова НАЗАД (заполняете пробелом позицию последнего введенного символа и уменьшаете значение указателя). Используя таким образом слово **KEY** в цикле, можно создавать любые интерпретаторы клавиатуры или редакторы. Последнее определение, ЦИФРЫ, демонстрирует использование слова EXPECT#. Итак, в листинге не осталось больше ничего, что было бы вам не известно, поэтому никаких причин для того, чтобы отложить рассмотрение этой программы, у вас нет. (Более подробную информацию вы найдете в [1].)

```
Block# 350
\ Ввод чисел   часть 1

: НАЗАД  8 EMIT ;
: ЗБ? ( c -- t=клавиша-забоя)  8 = ;   \ в некот.сист. 12
: ВК? ( c -- t=клавиша-RETURN) 13 = ;   \ в некот.сист. 0
: #?  ( c -- t=правильная-цифра)  ASCII 0  ASCII 9 1+ WITHIN ;

Block# 351
\ Ввод чисел   часть 2
: ПОЛУЧЕНИЕ   ( 1-й-адр посл-а+1 текущ-адр c -- 1-й-адр
                   посл текущ' )
   >R  2DUP > IF  R@ DUP EMIT  OVER C! 1+ THEN  R> DROP ;
: НАОБОРОТ ( 1-й-адр посл-а+1 текущ-адр c -- 1-й-адр посл
                   текущ' )
   DROP SWAP >R  2DUP < IF  НАЗАД SPACE НАЗАД  1- DUP 1 BLANK
   THEN  R> SWAP ;
: EXPECT#  ( a макс-длина -- факт.-длина)
   OVER +  OVER  BEGIN  KEY  DUP BK?  NOT WHILE
       DUP #?  IF  ПОЛУЧЕНИЕ          ELSE
       DUP ЗБ? IF  НАОБОРОТ           ELSE  DROP
   THEN THEN  REPEATE  ROT 2DROP  SWAP - ;

: ЦИФРЫ  ( #цифр -- d)
   PAD SWAP  2DUP 1+ BLANK  EXPECT# DROP  PAD 1- NUMBER ;
```

СРАВНЕНИЕ СТРОК

Для сравнения строк предусмотрены следующие два слова Форта:

-TEXT	(a1 # a2 -- ?)	Сравнение двух строк длиной # , начинающихся с a1 и a2. Если сравнение успешное, на стек заносится ложь. Если нет, на стек заносится истина (положительное число, если двоичное представление строки1 > двоичного представления строки2 , и отрицательное, если стр.1 < стр.2).
-MATCH	(d # s # -- a ?)	Поиск фрагмента длиной # ,начинающегося с адреса s (источник) в области памяти длиной # , начинающейся с адреса d (получатель). Если поиск завершился успешно, на стек помещается начало искомого фрагмента в заданной области памяти и ложь. В противном случае неправильный адрес и истина.

С помощью слова — **TEXT** вы можете либо сравнить две строки, либо проверить порядок их расположения (по алфавиту)[1]. В гл. 12 приводится пример использования слова—**TEXT** для выявления полного совпадения строк.

Так как для быстроты слово—**TEXT** осуществляет сравнение поячеечно, необходимо следить за тем, чтобы при наличии машин с ячеечной адресацией слову—**TEXT** выдавались адреса, выравненные по границе ячейки. Например, если вы хотите сравнить вводимую строку со строкой, находящейся в каком-то массиве, перенесите вводимую строку в рабочую область (посредством—**TEXT**, а не **WORD**), поскольку адрес PAD лежит на границе ячейки. Аналогичным образом, когда вам нужно проверить некую строку, находящуюся в блочном буфере, убедитесь в том, что ее адрес выравнен по границе ячейки, или, если вы не можете этого сделать, перед выполнением проверки перешлите строку по выравненному адресу (используя **CMOVE**).

Отметим, что дефис в слове —**TEXT**, как и символ « ¬» кода ASCII, означает логическое «нет». По этой причине указанный префикс удобно применять в именах слов, помещающих в стек флаги с противоположным логическим значением (т. е. нуль представляет истину, а ненулевое значение — ложь).

Если в вашей системе нет слова —**TEXT**, вы можете загрузить приведенное ниже определение. Конечно, для быстроты определение слова — **TEXT** обычно пишется в машинных кодах.

```
: -TEXT ( a1 # a2 -- f=сравнение  ! полож=1>2 ! отр=1<2 )
   2DUP + SWAP  DO  DROP 2+  DUP 2- @
      I @ -  DUP  IF DUP ABS / LEAVE  THEN
   2 +LOOP  SWAP DROP ;
```

---

[1] *Для пользующихся процессорами* INTEL, DEC и Zilog. Для того чтобы выполнить такую проверку, вы должны расположить байты в обратном порядке.

Слова —**MATCH** применяется в командах редактирования, таких, как **F** и **S**, по которым должен осуществляться поиск некоторого фрагмента в памяти, содержащей данный фрагмент. Как и в случае с —**TEXT**, желательно, чтобы слово —**MATCH** было написано в машинных кодах. Если этого сделать не удается, можете воспользоваться следующим определением высокого уровня, которое вам подойдет (для надежности и переносимости описанные далее слова не используют такие приемы, как принудительный выход из циклов посредством **EXIT**, что ускорило бы их выполнение):

```
VARIABLE 'ИСТОЧНИК    ( адрес исходного фрагмента)
VARIABLE ИСТОЧНИК#    ( длина исходного фрагмента)
VARIABLE ФЛАГ         ( t=сравнения-не-призошло)
: -MATCH   ( d # s # -- a t=сравнения-не-произошло)
   SWAP 'ИСТОЧНИК !  DUP ИСТОЧНИК# !  - DUP 0< NOT IF
      1+ 0 DO  0 ФЛАГ !
         'ИСТОЧНИК @  ИСТОЧНИК# @  0 DO
           OVER I + C@  OVER I C@ - IF -1 ФЛАГ ! LEAVE THEN
         LOOP  DROP
      ФЛАГ @ 0= IF  LEAVE  THEN  1+  LOOP
   ФЛАГ @  THEN ;
```

СТРОКОВЫЕ ЛИТЕРАЛЫ

Текстовая строка, скомпилированная в словарь со средствами получения ее адреса, называется *строковым литералом*. Общим словом для создания строковых литералов является слово **STRING** (СТРОКА)[1]. Оно функционирует следующим образом:

```
CREATE СООБЩЕНИЕ  BL STRING ПРИВЕТ
СООБЩЕНИЕ COUNT TYPE ПРИВЕТ ok
```

Слово **STRING** выбирает из стека в качестве аргумента символ в коде ASCII — ограничитель строки — и считывает до него фрагмент из входного потока, компилируя его в словарь как строку со счетчиком.

Вы можете воспользоваться словом **STRING** для создания строковых массивов. Вспомните наше определение слова МАРКИ-

[1] *Для пользователей всех систем, кроме полиФорта.* Во многих системах (но не всюду) вы можете воспользоваться следующим определением:

```
: STRING  ( c -- )  WORD C@ 1+ ALLOT ;
```

Мы полагаем, что в таких системах буфер слова **WORD** начинается с **HERE**, поэтому здесь не требуется пересылка строки и счетчика в словарь. Адрес, оставляемый в стеке словом **WORD**, является адресом со счетчиком, так что слово **C@** помещает значение этого счетчика в вершину стека. Затем слово **ALLOT** продвигает указатель словаря по длине строки (плюс один из-за счетчика), резервируя память для нее в словаре.

РОВКА в программе сортировки яиц, где мы применяли вложенные конструкции **IF THEN**. На этот раз сделаем сообщения о категории яиц одинакового размера (по восемь букв) и соединим их в одну строку с помощью единственного строкового литерала:

```
CREATE "МАРКИРОВКА"
ASCII " STRING Брак    Мелкие  Средние Крупные Оч.крупнОшибка"
```

Обращаясь к слову МАРКИРОВКА, мы получаем адрес данной строки. Можно вывести любое сообщение о категории яиц, вычислив смещение внутри строки. Например, если нужно выдать сообщение о категории 2, то добавляем единицу (чтобы проскочить байт со счетчиком), затем прибавляем 16(2*8) и выводим восемь символов имени:

```
"МАРКИРОВКА" 1+  16 +  8 TYPE
```

Теперь переопределим слово МАРКИРОВКА так, чтобы оно выбирало из стека номер категории от нуля до пяти и использовало его как индекс в нашем строковом массиве:

```
: МАРКИРОВКА  ( категория# -- )
   8 *  "МАРКИРОВКА" +  8 TYPE SPACE ;
```

Строковый массив такого вида иногда называют *суперстрокой*. По соглашению об именовании имя суперстроки должно быть заключено в кавычки.

Новый вариант слова МАРКИРОВКА выполняется немного быстрее, потому что ему не требуется производить ряд операций сравнения до тех пор, пока не будет найдена соответствующая категория. Адрес выводимого сообщения вычисляется по аргументу. Но если аргумент слова МАРКИРОВКА выходит за пределы диапазона от нуля до пяти, то выведется «мусор». В том случае, когда слово МАРКИРОВКА используется только внутри слова РАЗМЕР-ЯИЦ, проблем еще нет, однако если вы собираетесь отдать это слово конечному пользователю (т. е. оператору), следует обеспечить контроль:

```
: МАРКИРОВКА  0 MAX  5 MIN  МАРКИРОВКА ;
```

Вы встретитесь со словом **STRING** снова при рассмотрении определения слова **.**" в гл. 11.

Во многих Форт-системах существует еще одно слово для создания строковых литералов. Это слово не имеет стандартного имени, поэтому присвоим ему имя **LIT**". Его можно употреблять только внутри определений. Оно подобно слову **.**", но вместо выво-

да строки оставляет в вершине стека адрес строки со счетчиком. По существу, выражение

```
: 1TECT   LIT" Что случилось?" COUNT TYPE ;
```

эквивалентно выражению

```
: 2TECT   ." Что случилось?" ;
```

Слово **LIT"** является более мощным, чем **STRING** (поэтому менее употребительным). Последнее может применяться для определения слова **LIT"**.

В системе-83, созданной Лэксеном и Перри, используется слово ", которое аналогично слову **LIT"**, но оставляет в вершине стека адрес и значение счетчика. По нашему мнению, их вариант разбиения не обеспечивает должной гибкости, поэтому обратимся к варианту **LIT"**. Если вы будете знать, чем отличаются упомянутые варианты, их реализация не вызовет у вас никаких затруднений.

Ниже приводится перечень слов Форта, рассмотренных в настоящей главе.

UPDATE	(--)	Блок, доступ к которому осуществлялся в последнюю очередь, отмечается как модифицируемый. Этот блок будет впоследствии переписан во внешнюю память, если его буфер потребуется для размещения другого блока или в результате выполнения слова FLUSH.
SAVE-BUFFERS	(--)	Запись содержимого всех обновленных буферов в сответствующие блоки внешней памяти. У всех буферов погашается признак обновления, но они продолжают оставаться распределенными.
FLUSH	(--)	Осуществляется SAVE-BUFFERS, затем происходит погашение признака обновления всех буферов. Используется при установке или смене накопителей внешней памяти.
EMPTY-BUFFERS	(--)	Все блочные буфера отмечаются как пустые независимо от их содержания. Обновленные блоки во внешнюю память не записываются.
BLOCK	(u -- a)	На стек заносится адрес первого байта в блоке u . Если данного блока еще в памяти нет, то происходит его пересылка из внешней памяти в тот буфер, к которому дольше всех не было доступа. Если блок, занимающий данный буфер, обновлялся (то есть был модифицирован), то перед считыванием блока u в буфер содержимое последнего будет переписано во внешнюю память.
BUFFER	(u -- a)	Функции те же, что и у BLOCK, за исключением того, что сам блок из внешней памяти не считывается.

TYPE	(a # --)	Происходит выдача заданного количества символов, начиная с заданного адреса, на текущее внешнее устройство
-TRAILING	(a u1 -- a u2)	Удаление незначащих пробелов из строки с заданным адресом путем уменьшения значения счетчика от u1 (счетчик исходных байтов) до u2 (счетчик байтов, полученных в результате вычеркивания пробелов)
>TYPE	(a # --)	То же самое, что и TYPE, за исключением того, что перед выдачей выводимая строка помещается в pad Используется в мультипрограммных системах для вывода строк, находящихся в блоках на диске.
MOVE	(a1 a2 u --)	Поячеечное копирование участка памяти длиной в u байтов, начинающегося с адреса a1, в участок памяти, начинающийся с a2. Пересылка идет с a1 в сторону увеличения адресов.
CMOVE	(a1 a2 u --)	Побайтное копирование участка памяти длиной u байтов, начинающегося с a1, в участок памяти, начинающийся с a2. Пересылка идет с a1 в сторону увеличения адресов.
CMOVE>	(a1 a2 u --)	Копирование участка памяти длиной u байтов, начинающегося с адреса a1, в участок памяти, начинающийся с адреса a2. Копирование начинается с КОНЦА строки и продвигается в сторону уменьшения адресов.
BLANK	(a u --)	Заполнение участка памяти длиной u байт символом пробела в коде ASCII.
KEY	(-- c)	Занесение на стек значения в коде ASCII очередного доступного символа на текущем устройстве ввода
EXPECT	(a u --)	Ожидание u символов (или нажатий клавиши "RETURN") с клавиатуры и запоминание их в участок памяти, начинающийся с адреса a и продолжающийся в сторону увеличения адресов. На нажатие клавиши "ЗАБОЙ" осуществляется возврат курсора
SPAN	(-- a)	Содержится количество символов, полученных словом EXPECT
WORD	(c -- a)	Чтение слова, ограниченного заданным символом, из входного потока. Полученный фрагмент оформляется в виде строки со счетчиком и ее адрес помещается в стек.
COUNT	(a -- a+1 #)	Преобразование адреса строки со счетчиком (длина которой находится в первом строки) в формат, соответствующий использованию словом TYPE, а именно: в стек заносится адрес начала текста строки и ее длина.
>IN	(-- a)	Пользовательская переменная, содержащая смещение обрабатываемого символа относительно начала входного потока.

BLK	(-- a)	Пользовательская переменная, содержащая номер блока во внешней памяти, интерпретируемого в качестве входного потока. Если в BLK содержится нуль, то входной поток поступает из буфера входного текста.
CONVERT	(ud1 a1 -- ud2 a2)	Начиная с адреса a1+1 (байт, содержащий длину, пропускается), CONVERT преобразует строку в двоичное значение, которое зависит от текущей системы счисления (значения BASE). Полученное значение накапливается в ud1 и остается на стеке как ud2. Процесс продолжается до тех пор, пока не встретится символ, который не может быть истолкован как цифра в текущей системе счисления. Адрес этого символа заносится на стек как a2.
NUMBER	(a -- d)	Преобразование текста, начинающегося с адреса a+1, в двоичное значение с учетом текущей системы счисления (значения BASE). Строка может предваряться знаком минус, что делает полученное значение отрицательным.
-TEXT	(a1 # a2 -- ?)	Сравнение двух строк длиной # , начинающихся с a1 и a2. Если сравнение успешное, на стек заносится ложь. Если нет, на стек заносится истина (положительное число, если двоичное представление строки1 > двоичного представления строки2 , и отрицательное, если стр.1 < стр.2).
-MATCH	(d # s # -- a ?)	Поиск фрагмента длиной # ,начинающегося с адреса s (источник) в области памяти длиной # , начинающейся с адреса d (получатель). Если поиск завершился успешно, на стек помещается начало искомого фрагмента в заданной области памяти и ложь. В противном случае неправильный адрес и истина.
STRING	(c --)	Компиляция строкового литерала, ограниченного символом c, в словарь как строки со счетчиком.
LIT" xxx"	период-выполнения: (-- a)	Компиляция строкового литерала xxx, ограниченного двойной кавычкой. В период выполнения на стек помещается адрес строки со счетчиком. Используется только внутри определений.

ОСНОВНЫЕ ТЕРМИНЫ

Виртуальная память. Использование внешней памяти (например, диска), так как если бы она была оперативной, включая те средства операционной системы, которые обеспечивают эту возможность.

Ожидание. Приостановка исполнения программы для ожидания ввода с клавиатуры (в противоположность сканированию).

Относительный указатель. Переменная, определяющая абсолютный адрес некоторого участка, а его расположение относительно начала массива или строки.

Сканирование. «Заглядывание» вперед по входному потоку с целью просмотра оставшегося текста или поиска фрагмента, ограниченного заданным символом.

Суперстрока. В Форте — это массив символов, содержащий ряд строк. Доступ к любой из них может быть получен посредством индексирования этого массива.

Упрятывание информации. Прием, используемый программистом: он локализует информацию о том, как работает конкретная конструкция (особенно в тех случаях, когда что-то может измениться в последующих версиях или будет повторно применяться) в пределах некоторой программы или комплекса программ.

УПРАЖНЕНИЯ

10.1. Введите несколько известных цитат в некоторый доступный блок, скажем в блок 228. Определите слово с именем ЗАМЕНА, которое берет из стека два значения в коде ASCII и заменяет все вхождения первого символа в блоке 228 на второй символ. Так следующее выражение заменит все литеры А литерами Е:

65 69 ЗАМЕНА

10.2. Определите слово с именем ПРОГНОЗ, которое будет выдавать на вашем терминале некоторый прогноз, например «Вы получите хорошие новости по почте». Прогноз должен выбираться случайным образом из списка, включающего 16 (или менее) вариантов. Каждый прогноз может иметь длину до 64 символов.

10.3. а) Определите слово ДА/НЕТ?, которое можно использовать в прикладной программе, требующей утвердительного или отрицательного ответа пользователя. Определение должно ожидать нажатия одной клавиши, после которого послать в стек значение истина, если была нажата клавиша Y, и ложь при нажатии любой другой клавиши.

б) Иногда пользователь нажимает эту клавишу, работая в режиме нижнего регистра. Перепишите определение ДА/НЕТ? таким образом, чтобы оно воспринимало нажатие клавиши Y в обоих регистрах как «да».

в) Пользователь может по ошибке вместо клавиши Y (например, при записи файла на диск, над которым он проработал последние три часа) нажать другую клавишу. Перепишите определение ДА/НЕТ? таким образом, чтобы оно воспринимало символы, поступающие только при нажатии клавиш Y и N в верхнем и нижнем регистрах и не прекращало своего выполнения до тех пор, пока не получит сигналы от этих клавиш с записью в стек истины в случае Y и лжи в случае N.

10.4. Согласно восточной легенде Будда наделял всех людей, родившихся в определенном году, чертами, присущими одному из 12 животных. На этой основе построен циклический календарь. Цикл составляет 12 лет, по истечении которых он возобновляется. Например, считается, что 1900 год был «годом крысы». Внутри цикла годы, обозначенные названием животного, чередуются в следующем порядке: год крысы, год быка, год тигра, кролика, дракона, змеи, лошади, барана, обезьяны, петуха, собаки, свиньи. Напишите слово с именем ЖИВОТНОЕ, которое бы выводило название конкретного животного в зависимости от его места в приведенном списке, например:

```
0 .ЖИВОТНОЕ КРЫСА ok
```

Далее напишите слово с именем (ГОРОСКОП), отражающим искусство составления гороскопа для рожденных под тем или иным циклическим знаком так, чтобы оно могло брать в качестве аргумента год рождения и выводить название соответствующего животного.

Наконец, напишите слово с именением ГОРОСКОП, которое запрашивало бы

у пользователя год рождения. При этом на экран должны выводиться четыре символа подчеркивания, отмечая поле для ввода года рождения, после чего должен отработать четыре раза «ЗАБОЙ», возвращая курсор в первоначальное положение. Используйте определение слова EXPECT#, введенное в настоящей главе («Построение программы ввода чисел с помощью слова KEY».) После нажатия клавиши RETURN на экране должно появиться название соответствующего животного по циклическому календарю.

10.5. В настоящей главе мы определили слово «добавить» для ввода в базу данных бессмысленных фраз. Этот вариант позволяет по каждому такому слову вводить только один фрагмент. Переопределите слово «добавить», чтобы оно выбирало из входного потока по три поля, каждое из которых представляет по одному столбцу и отделяется запятыми, например:

```
начало
добавить высокий, культурный, уровень
```

При следующем использовании слова «добавить» заполняется очередная строка базы данных.

10.6. Мы полагаем, что слово CONVERT может применяться при анализе числа, представленного, скажем, таким замысловатым способом: «6/20/88». Напишите определение с именем >ДАТА, которое сканировало бы подобные числа по заданному адресу и оставляло в вершине стека дату в виде двух 16-разрядных чисел. При этом старшая ячейка содержала бы год (1988), а младшая — месяц и день, причем месяц бы находился в старшем байте. Затем создайте слово СКАНИРОВАНИЕ-ДАТЫ, которое сканировало бы входной поток в поисках даты и оставляло бы в вершине стека то же значение, что и слово >ДАТА.

10.7. В гл. 8 было показано, как создаются массивы ячеек (массив из 16-разрядных чисел) в словаре. В данном упражнении нужно создать *виртуальный массив* (не в оперативной памяти, а на диске), состоящий из 16-разрядных значений. Сначала найдите три доступных смежных блока, где вы будете хранить свой массив. В блоках не должно быть никакой текстовой информации, так как в массиве придется хранить двоичные числа. Определите переменную, которая указывала бы на первый из трех блоков. Далее создайте слово с именем ЭЛЕМЕНТ со следующей стековой нотацией:

```
( индекс -- адрес)
```

Это слово переводит индекс ячейки в абсолютный адрес внутри блочного буфера. Обратите внимание на то, что в первом блоке будут храниться только 512 ячеек. Кроме того, оставляемый в вершине стека адрес указывает на второй блок набора и т. д. Слово ЭЛЕМЕНТ также должно вызывать UPDATE.

Теперь созданное вами нужно проверить. Напишите программу инициализации первых 600 ячеек некоторыми значениями и выведите их содержимое на экран, после чего переопределите слово ЭЛЕМЕНТ так, чтобы первый элемент массива резервировался бы для хранения счетчика запомненных элементов. Кроме того, определите слово ИСПОЛЬЗОВАНО для занесения в стек значения счетчика. Напишите программу очищения массива. Определите слово ПОМЕСТИТЬ, которое выбирает из стека 16-разрядное значение и добавляет его к массиву на первое свободное место.

Далее определите слово ВВОД, которое добавляет к массиву два 16-разрядных значения, взятых из стека, причем значение, взятое вторым, добавляется первым. И наконец, определите слово ТАБЛИЦА для вывода информации на экран: шесть чисел на строку. (Благодарим за пример К. Хэрриса.)

ЛИТЕРАТУРА

1. Ham, Michael, "Think Like a User, Write Like a Fox," *Forth Dimensions*, VI/3, p. 23.

Глава 11

РАСШИРЕНИЕ КОМПИЛЯТОРА: ОПРЕДЕЛЯЮЩИЕ И КОМПИЛИРУЮЩИЕ СЛОВА

В этой главе мы переходим на следующую ступень изучения Форта. В предыдущих главах рассматривались такие его свойства, которые могут быть присущи любому языку программирования. Вы познакомились с арифметическими операциями, управляющими структурами, структурами данных, вводом-выводом, внешней памятью и т. д. Даже двоеточие имеет аналогии во многих языках в виде процедурного аппарата. Здесь же вы столкнетесь с совершенно иным уровнем возможностей языка: вы научитесь расширять сам компилятор с Форта. Создание новых определяющих и компилирующих слов служит основой для разработки исключительно читабельных, легко сопровождаемых программ и, вероятно, представляет собой самое значительное из средств Форта.

ЧТО ТАКОЕ ОПРЕДЕЛЯЮЩЕЕ СЛОВО?

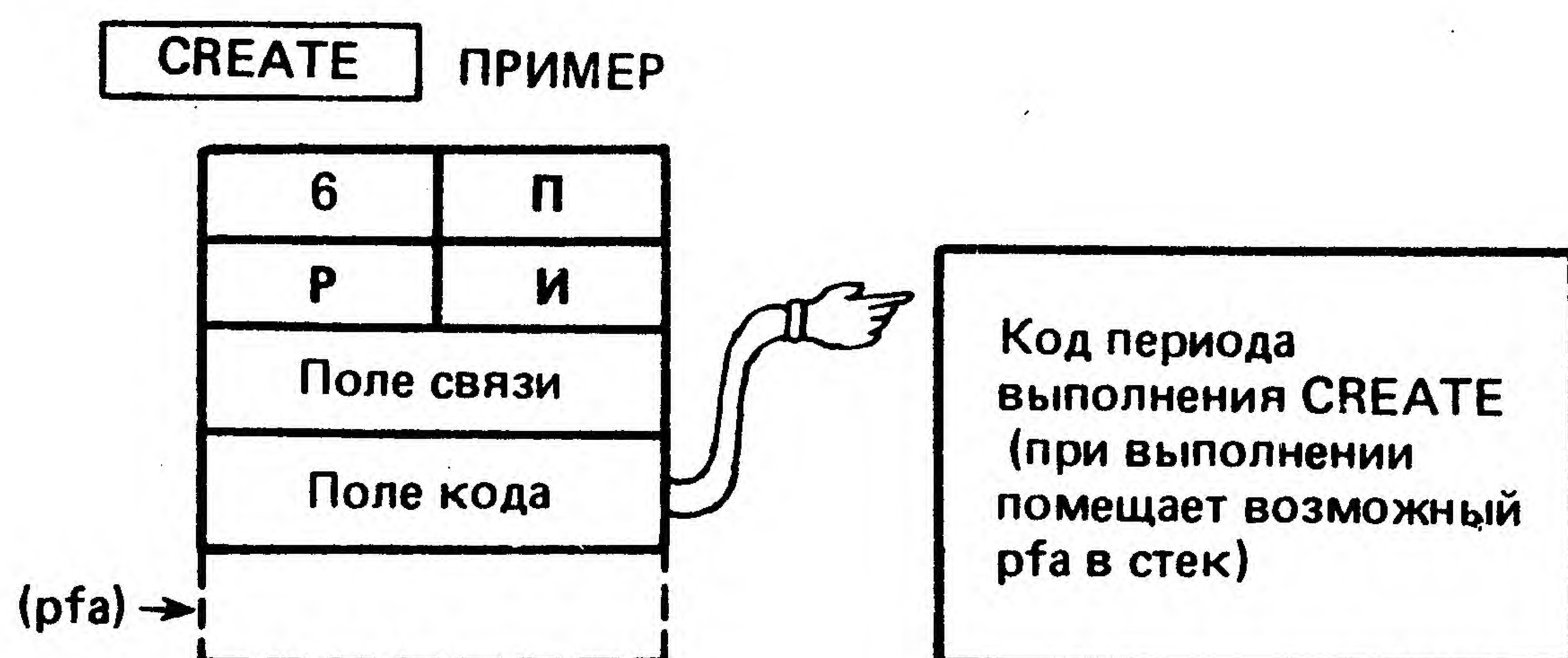
Любое слово, которое создает новый заголовок в словаре, является определяющим. Некоторые из определяющих слов вы уже знаете, в частности

```
:
VARIABLE
CONSTANT
CREATE
```

Все они обладают одним общим свойством: «определять» слова и добавляют новые имена к словарю. В отличие от других языков Форт позволяет создавать свои собственные определяющие слова. Для чего это нужно? Вообще говоря, определяющие слова способствуют хорошему разбиению программы. Мы уже неоднократно излагали вам концепцию разбиения (см. гл. 8). Такая программа легко читается, легко воспринимается и легко исправляется.

Использование определяющих слов способствует хорошему разбиению, потому что дает возможность создавать целые *классы*, или *семейства*, слов с похожими свойствами. Признаки, объединяющие члены некоторого семейства, задаются не в определении каждого члена, а в определяющем слове.

Прежде чем предложить вам пример, рассмотрим, как специфицируются определяющие слова. Основой всех определяющих слов является простейшее из них — слово **CREATE**[1]. Это слово выбирает из входного потока имя и создает для него в словаре заголовок.

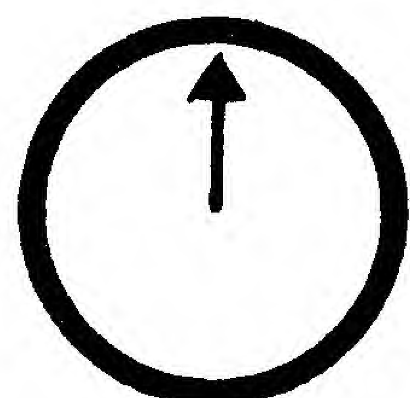


Слово **CREATE** считается родителем, а слово ПРИМЕР — ребенком. Что делает ребенок в период выполнения? Он помещает свой собственный pfa в стек. А откуда он знает, что нужно делать именно это? Мы не задавали непосредственно ему никакой программы для выполнения. Ответ прост — ПРИМЕР вообще не содержит кода периода выполнения. Его указатель кода указывает на родителя (**CREATE**), у которого код периода выполнения есть.

Предположим, что в Форте нет слова **VARIABLE**. Мы можем его определить:

```
: VARIABLE   CREATE  0 , ;
```

Мы обратились к слову **CREATE** внутри определения через двоеточие. Что при этом произойдет? Давайте проследим за выполнением в хронологическом порядке:



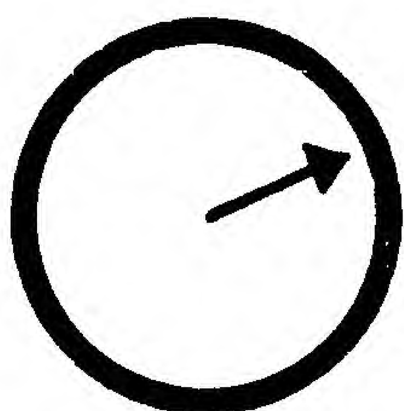
Фаза 1

```
: VARIABLE   CREATE  0 , ;
```

[1] *Для пользователей систем фиг-Форта.* Не забудьте переопределить слово **CREATE** следующим образом:

```
: CREATE   <BUILDS DOES> ;
```

Определяется определяющее слово **VARIABLE**



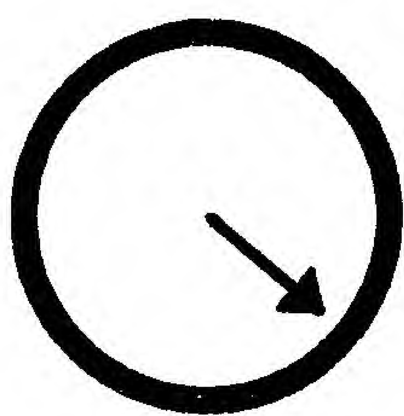
Фаза 2

```
VARIABLE АПЕЛЬСИНЫ
```

Исполняется слово **VARIABLE**, в свою очередь выполняя две функции:

· **GREATE**. Создает с помощью **GREATE** заголовок в словаре с именем АПЕЛЬСИНЫ и указателем кода, который ссылается на код периода выполнения слова **GREATE**;

0, Засылает 16-разрядный нуль в поле параметров вновь созданной переменной и выделяет ячейку памяти.



Фаза 3

```
АПЕЛЬСИНЫ
```

Исполняется слово АПЕЛЬСИНЫ. Так как указатель кода слова АПЕЛЬСИНЫ ссылается на код периода выполнения **CREATE**, pfa этого слова помещается в вершину стека. Конечно, мы могли бы обойтись без слова **VARIABLE**. Вполне достаточно ввести следующее:

```
CREATE ПРИМЕР  0 ,
```

Однако такая запись менее изящна, поскольку здесь разбиты на отдельные действия создание заголовка и выделение памяти.

Пример с определением слова **VARIABLE** демонстрирует лишь половину возможностей механизма определяющих слов. Если бы мы вместо **VARIABLE** воспользовались словом **CREATE**, то нам пришлось бы подкорректировать единственное место — в фазе 1, где происходит определение слова АПЕЛЬСИНЫ. И напротив, в фазе 3 слово АПЕЛЬСИНЫ вело бы себя одинаково при определении посредством как **CREATE**, так и **VARIABLE**.

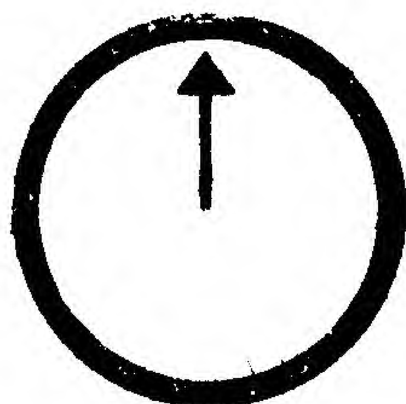
Кроме того, Форт дает возможность создавать определяющие слова-родители, задающие поведение своих детей во время исполнения. Ниже в качестве примера приводится правильное определение слова **CONSTANT** (хотя на самом деле слова, подобные **VARIABLE** и **CONSTANT**, обычно определяются с помощью машинных кодов):

```
: CONSTANT   CREATE ,   DOES> @ ;
```

Здесь «собака зарыта» в выполнении слова **DOES>**, которое отмечает начало кода периода выполнения для всех своих детей. Как вы знаете, константы (т. е. дети определяющего слова **CONSTANT**) засылают в стек свои значения, которые хранятся в их поле параметров. Поэтому слово @, которое следует за **DOES>**, выбирает значение константы из ее собственного pfa.

В любом определяющем слове **CREATE** отмечает начало действий, выполняемых в период компиляции (фаза 2), а **DOES>** — конец действий периода компиляции и начало операций периода выполнения (фаза 3).

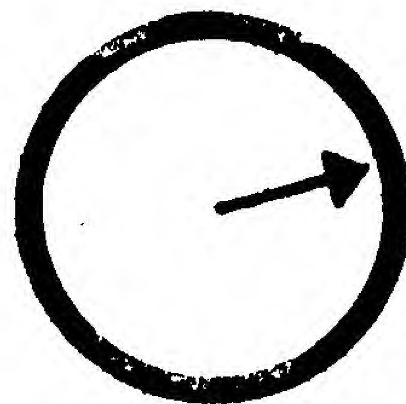
Проследим еще раз все наши действия:



Фаза 1

```
: CONSTANT   CREATE ,   DOES> @ ;
```

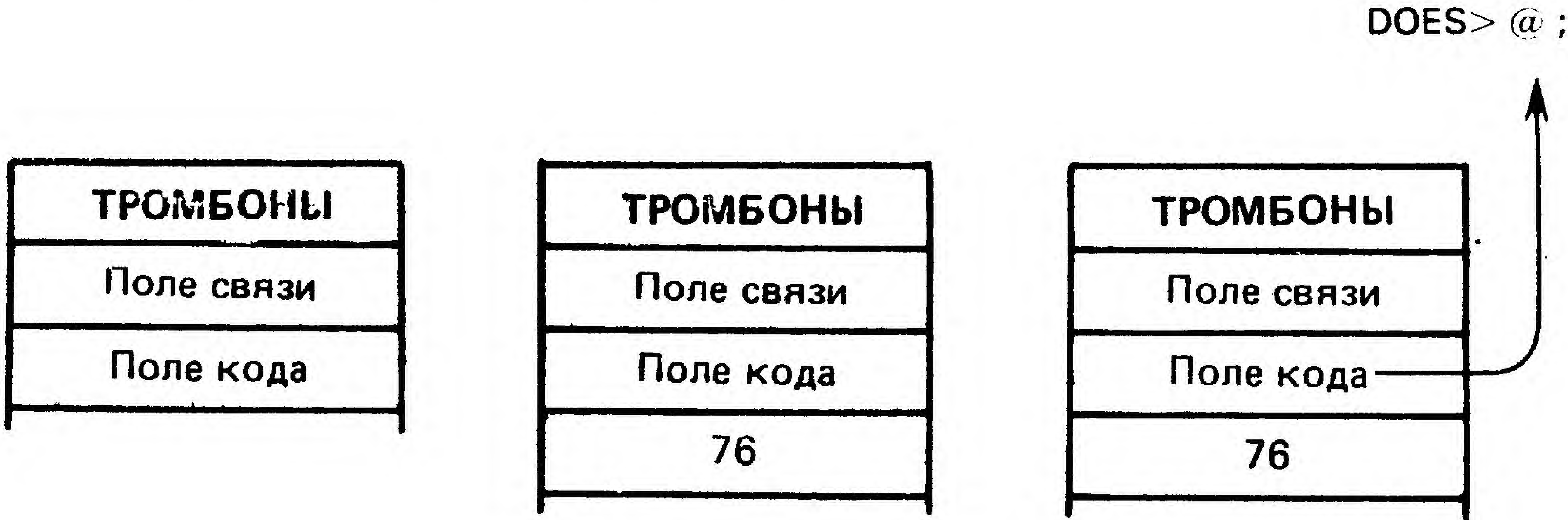
Определение определяющего слова **CONSTANT**.



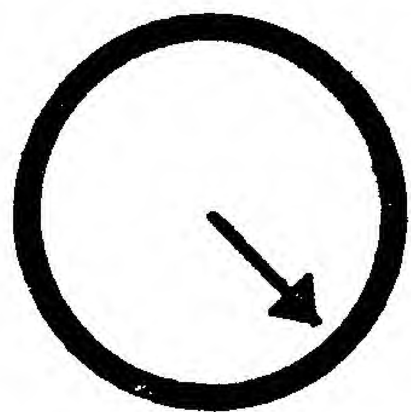
Фаза 2

```
76 CONSTANT ТРОМБОНЫ
```

Исполнение слова **CONSTANT**, которое в свою очередь выполняет следующие три действия:



Создает с помощью **CREATE** заголовок словарной статьи с именем ТРОМБОНЫ. Выбирает из стека значение (например, 76) и заносит его в поле параметров константы. Устанавливает указатель поля кода слова ТРОМБОНЫ на код, следующий за словом **DOES**.



Фаза 3

ТРОМБОНЫ

Выполнение слова ТРОМБОНЫ. Поскольку указатель поля кода слова ТРОМБОНЫ теперь указывает код, следующий за **DOES>**, выбирается значение (76) и помещается в вершину стека. Обратите внимание на то, что в фазе 3 слово **DOES>** сначала помещает в вершину стека pfa ребенка. Иными словами, определение

```
: VARIABLE   CREATE  0 , ;
```

эквивалентно следующему:

```
: VARIABLE   CREATE  0 , DOES> ;
```

Последнее в период выполнения помещает в вершину стека pfa, а в период компиляции ничего не выполняет.

Так как определяющее слово задает поведение в двух различных фазах (периодах компиляции и выполнения), нужно соответствующим образом отразить это в стековой нотации. В частности, стековый комментарий определения слова **CONSTANT** имеет вид:

```
: CONSTANT  ( n -- )  CREATE ,
   DOES>  ( -- n)  @ ;
```

Верхняя строка стекового комментария описывает поведение родителя во время компиляции, а нижняя, после **DOES>**, задает поведение ребенка.

DOES>	период-выполнения: (-- a)	Используется при создании определяющих слов. Отмечается конец участка периода компиляции и начала участка периода выполнения. Операции периода выполнения определены на высокоуровневом Форте. Во время выполнения на стеке будет находиться pfa определенного слова.

ОПРЕДЕЛЯЮЩИЕ СЛОВА ВЫ МОЖЕТЕ СПЕЦИФИЦИРОВАТЬ САМИ

Рассмотрим три класса определяющих слов. Слова первого класса дают возможность выделять фрагменты с похожими свойствами из серии определений через двоеточие. В предыдущей главе вы имели дело со следующими определениями:

```
: 1ПРИЛАГАТЕЛЬНОЕ  10 CHOOSE  0 .БРЕД ;
: 2ПРИЛАГАТЕЛЬНОЕ  10 CHOOSE  1 .БРЕД ;
: СУЩЕСТВИТЕЛЬНОЕ  10 CHOOSE  2 .БРЕД ;
```

Если бы вам пришлось определять и другие части речи, то можно было бы их выделить в отдельное слово «часть-речи»:

```
: ЧАСТЬ  ( столбец# -- )  10 CHOOSE  SWAP .БРЕД ;
: 1ПРИЛАГАТЕЛЬНОЕ   0 ЧАСТЬ ;
: 2ПРИЛАГАТЕЛЬНОЕ   1 ЧАСТЬ ;
: СУЩЕСТВИТЕЛЬНОЕ   2 ЧАСТЬ ;
```

Однако при этом слишком расточительно расходуется память. Поскольку различие между показанными тремя словами заключено только в одном числе, их незачем определять через двоеточие. Целесообразнее воспользоваться таким приемом:

```
: ЧАСТЬ  ( столбец# -- )  CREATE ,
   DOES> @  10 CHOOSE  SWAP .БРЕД ;
0 ЧАСТЬ 1ПРИЛАГАТЕЛЬНОЕ  1 ЧАСТЬ 2ПРИЛАГАТЕЛЬНОЕ
2 ЧАСТЬ СУЩЕСТВИТЕЛЬНОЕ
```

Приведенное определение слова ЧАСТЬ аналогично определению слова **CONSTANT**[1], с той лишь разницей, что в период выполнения помимо занесения в стек номера столбца оно также определяет номер строки посредством датчика случайных чисел и обращается к слову .БРЕД.

Следующий довольно большой класс определяющих слов позволяет выделять повторяющиеся фрагменты кодов периода компиляции. Допустим, требуется назначить ряд блоков для группы студентов начиная с блока 360, причем каждому студенту нужно выделить по 25 блоков. Для этого понадобится несколько кон-

---

[1] *Для пуристов.* Иногда имеет смысл объединять существующие определяющие слова, если они выполняют большую часть тех функций (или все), которые должны выполняться во время компиляции. Так, выражение **CREATE**, с тем же эффектом можно заменить словом **CONSTANT**. Стандарт, однако, не рекомендует пользоваться такими приемами и не во всех системах это получится.

стант, обозначающих номер начального блока для каждого студента. Константы можно определить так:

```
360 CONSTANT ВАСИЛЬЕВ
385 CONSTANT СИМОНЧИК
410 CONSTANT РЯБИНИНА
435 CONSTANT ИВАНОВА
460 CONSTANT ПЕТРОВА
485 CONSTANT ДЕМИН
```

Если у вас со сложением дела обстоят плохо, то вы обязательно ошибетесь. Кроме того, вдруг вы измените свое решение и выделите каждому студенту по 30 блоков? Есть более изящный прием. Он состоит в том, чтобы выделить вычисления во время компиляции в отдельное слово:

```
: +СТУДЕНТ  ( n -- n+25 n )  DUP  25 +  SWAP ;
     360  \  начало участка памяти, отведенной студентам
+СТУДЕНТ CONSTANT ВАСИЛЬЕВ      +СТУДЕНТ CONSTANT СИМОНЧИК
+СТУДЕНТ CONSTANT РЯБИНИНА      +СТУДЕНТ CONSTANT ИВАНОВА
+СТУДЕНТ CONSTANT ПЕТРОВА       +СТУДЕНТ CONSTANT ДЕМИН
.  .( Конец участка памяти, отведенного студентам ) CR
```

Последняя точка выбирает из стека номер блока.

Но и этот прием блекнет перед методом использования определяющих слов. Убедитесь сами:

```
: СТУДЕНТ  ( n -- n+25)  CREATE  DUP ,  25 +
   DOES>  ( -- n) @ ;
360  \  Начало участка, отведенного студентам
СТУДЕНТ ВАСИЛЬЕВ     СТУДЕНТ СИМОНЧИК     СТУДЕНТ РЯБИНИНА
СТУДЕНТ ИВАНОВА      СТУДЕНТ ПЕТРОВА      СТУДЕНТ ДЕМИН
.  .( Конец участка, отведенного студентам ) CR
```

Определяющее слово СТУДЕНТ и создает «константы», и управляет всеми вычислениями во время компиляции.

Определяющие слова третьего класса дают возможность создавать новые структуры данных, например многомерные массивы. Нас иногда упрекают в том, что структуры данных Форта бедны. Однако редко бывает так, чтобы какой-то набор структур данных удовлетворял всем прикладным областям. Форт же предоставляет удобные инструменты для создания определяющих слов привычных нам структур данных. Мы можем образовать структуры данных, которые во время выполнения осуществляют, помимо всего прочего, присущие только им операции. В информатике такие структуры называются *абстрактными типами данных*.

Рассмотрим два примера. Первый из них, более простой, демонстрирует определяющее слово, организующее одномерный мас-

сив. Можно, конечно, создать такой массив и без применения определяющего слова:

```
CREATE КЛАПАНЫ  30 ALLOT  \  байтовый массив установки клапанов
: КЛАПАН  ( i -- a)  \  преобр. номера клапана в абсолютный адрес
   КЛАПАНЫ + ;
```

Здесь слово КЛАПАН вычисляет индекс массива КЛАПАНЫ. Например, при выполнении выражения

```
6 КЛАПАН C@
```

будет включен гидравлический клапан 6.

Приведенное выше решение приемлемо только в тех случаях, когда в программе требуется один или два таких массива. Если же массивов должно быть больше, использование определяющего слова упростит программирование.

```
: МАССИВ  ( #байтов -- )  \  определение одномерного массива байтов
   CREATE  ALLOT
   DOES>  ( i -- a)  + ;

30 МАССИВ КЛАПАН
6 КЛАПАН C@
```

Рассмотрим выполнение этого определения. В фазе 1 определяется слово МАССИВ. В фазе 2 исполняется МАССИВ, который в свою очередь обращается к слову **CREATE** (чтобы определить КЛАПАН) и слову **ALLOT** (для резервирования 30 байтов под массив). В фазе 3 исполняется слово КЛАПАН, инициируя код периода выполнения слова МАССИВ с добавлением индекса (6) к начальному адресу массива.

Если внести изменения в определение определяющего слова перед его повторной компиляцией, то тем самым можно изменить характеристики всех слов, входящих в данное семейство. Такая возможность значительно упрощает разработку программ. Например, когда нужно при описании массива заполнить его нулями, вы должны соответствующим образом создать определение МАССИВ. Сначала вы определяете слово, которое аналогично **ALLOT**, но «обнуляет» выделенный участок памяти:

```
: 0ALLOT  ( #байтов -- )  HERE OVER  ERASE  ALLOT ;
```

Затем подставляете в определение МАССИВ вместо **ALLOT** слово 0ALLOT:

```
МАССИВ  ( #байтов -- )  \  определение одномерного массива байтов
   CREATE  0ALLOT
   DOES>  ( i -- a)  + ;
```

Можно также выделить в отдельный фрагмент некоторые отладочные процедуры, необходимые при разработке программы, а затем удалить их, предварительно убедившись в том, что она работает правильно. Ниже приводится вариант слова МАССИВ, где во время выполнения проверяется, не вышел ли индекс массива за установленные границы.

```
: МАССИВ  ( #байтов)  CREATE DUP ,  ALLOT
    DOES> ( i -- a)  2DUP @ U< NOT  ABORT" Выход за границу " + 2+ ;
```

Это происходит следующим образом:

DUP , ALLOT	Компиляция счетчика и выделение заданного количества байтов.
DOES> 2DUP @	По заданному на стеке индексу во время выполнения вычисляется: (i pfa i #)
U< NOT	Проверка того, что индекс не меньше максимального значения, а именно: запомненного счетчика. Так как U< является операцией сравнения над значениями без знака, то отрицательные аргументы будут трактоваться как числа, выходящие за границу, и приводить к аварийному сообщению.
ABORT" Выход за границу "	Аварийное завершение при выходе числа за диапазон.
+ 2+	В противном случае сложение индекса с pfa и добавление числа 2 для пропуска ячейки, содержащей счетчик.

Существует еще один способ использования определяющих слов, который помогает при создании программ. Допустим, вы вдруг решаете, что все ваши массивы, определенные с помощью слова МАССИВ, слишком велики, чтобы хранить их в памяти компьютера, и должны быть помещены на диск. Единственное, что вы должны в таком случае сделать, переопределить фрагмент периода выполнения в слове МАССИВ. Это новое определение вычислит номер блока, в котором содержится заданный байт, считает блок посредством **BLOCK** в некоторый буфер и оставит в вершине стека адрес требуемого байта относительно начала буфера. Массив, определенный подобным образом, может храниться в нескольких последовательных блоках (с использованием тех же средств, что и в упр. 10.7).

Ниже приведен пример определяющего слова, которое создает двумерный массив байтов заданного размера[1]:

```
: МАТРИЦА  ( #строк #столбцов -- )  CREATE  OVER , * ALLOT
    DOES> ( строка столбец -- a)  DUP @  ROT *  + +  2+ ;
```

---

[1] *Для любителей оптимизации.* Этот вариант будет выполняться еще быстрее:

```
: МАТРИЦА  ( #строк #столбцов -- )
  OVER  CONSTANT  HERE 2+ ,  * ALLOT
  DOES> ( строка столбец -- a)  2@  ROT *  + + ;
```

Для того чтобы создать массив размером 4 × 4 байта, вы должны написать:

		Столбцы			
		0	1	2	3
	0				
Строки	1				
	2		?		
	3				

```
4 4 МАТРИЦА ТАБЛИЦА
```

Выбрать же, к примеру, байт в строке 2 и в столбце 1 можно следующим образом:

```
2 1 ТАБЛИЦА C@
```

Вот так кратко можно описать выполнение слова МАТРИЦА. Поскольку аппаратные средства компьютера позволяют хранить только одномерные массивы, второе измерение необходимо моделировать. Мы представляем себе, что наш массив выглядит следующим образом:

Столбец 0	1	2	3
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

а на самом деле в памяти машины он хранится в виде

Столбец 0				1				2				3			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Если вам требуется адрес байта, расположенного в строке 2 столбца 1, то вы можете умножить номер столбца (1) на число строк в каждом столбце (4), а затем прибавить номер строки (2). В результате получается, что вам нужен шестой байт машинного представления массива. Примерно такие вычисления делают в период выполнения элементы, составляющие слово МАТРИЦА. Но для того чтобы их производить, как вы можете заметить, любое со-

ставляющее слово должно «знать» число строк в каждом столбце конкретного массива. Для этих целей слово МАТРИЦА во время компиляции вносит число строк в столбце в начало массива.

Для любознательных приведем стековые эффекты фрагмента периода выполнения слова МАТРИЦА:

ОПЕРАЦИЯ	СОДЕРЖИМОЕ СТЕКА
	строка столбец pfa
DUP @	строка столбец pfa #строк
ROT	строка pfa #строк столбец
*	строка pfa индекс-столбца
+ +	адрес
2+	скорректированный-адрес

К вычисленному адресу необходимо добавить двойку, потому что первая ячейка нашего массива содержит число строк.

Предлагаем вашему вниманию еще один пример, который, может быть, не очень полезен, но весьма нагляден:

```
\  Шаблоны с использованием определяющих слов
: STAR   42 EMIT ;
: .РЯД   ( b -- )   \  вывод звездочки на каждый бит из байта
   CR  8 0 DO DUP 128 AND  IF STAR ELSE SPACE THEN
     2* LOOP DROP ;
: ФОРМА  ( b1 b2 b3 b4 b5 b6 b7 b8 -- )  \  определение формы из 8-строк
   CREATE  8 0 DO   C,  LOOP
   DOES>   DUP  7 + DO  I C@  .РЯД  -1 +LOOP  CR ;

\  Формы:
HEX
18 18 3C 5A 99 24 24 24 ФОРМА ЧЕЛОВЕК
81 42 24 18 18 24 42 81 ФОРМА КОНЬ
AA AA FE FE 38 38 38 FE ФОРМА ЗАМОК
DECIMAL
```

Слово .РЯД выводит строку, состоящую из звездочек и пробелов, где звездочка соответствует единице, а пробел — нулю в восьмиразрядном двоичном представлении числа, находящегося в вершине стека, например:

```
2 BASE ! ok
00111001 .РЯД
  ***  * ok
DECIMAL ok
```

Наше определяющее слово ФОРМА берет из стека восемь аргументов и определяет шаблон, который при своем выполнении

выводит решетку 8 × 8 элементов, соответствующую этим восьми аргументам:

```
ЧЕЛОВЕК
   **
   **
  ****
 * ** *
*  **  *
  *  *
  *  *
  *  *
ok
```

Итак, определяющие слова могут быть чрезвычайно полезным инструментом. Создавая новое определяющее слово, вы тем самым расширяете свой компилятор. Традиционные языки не обеспечивают такой гибкости, потому что они представляют собой жесткие готовые программы, которые предлагают вам в обязательном порядке конкретный набор операторов. Реальная помощь определяющих слов заключается в том, что их применение позволяет упростить вашу программу. При правильном их употреблении вы можете сократить время программирования, уменьшить размер программы и повысить ее читабельность. В следующем разделе мы приведем еще один способ расширения средств компилятора Форта.

ЧТО ТАКОЕ КОМПИЛИРУЮЩЕЕ СЛОВО?

Если обычные слова Форта появляются внутри определения через двоеточие, то они компилируются в словарь. Эти слова во время компиляции ведут себя пассивно. *Компилирующие слова* также появляются внутри определения через двоеточие, но в противоположность первым активно влияют на процесс компиляции. Такие слова, как **IF, THEN, BEGIN, REPEAT** и **."** являются компилирующими.

Создатели Форта проявили последовательность и в данном случае. Коль уж существует тенденция не включать в язык все возможные определяющие слова, то компилирующих операторов в самом языке немного. Возможность управлять компиляцией путем образования собственных компилирующих слов предоставляет вам такую свободу, какую не может обеспечить ни один из известных языков программирования. Это средство позволяет локализовать информацию внутри соответствующих определений (не рассредоточивая ее по всей программе), что упрощает написание программы, облегчает ее чтение, восприятие и сопровождение. Вероятно, ваша программа будет выглядеть более привлекательной, если в язык включить оператор выбора вариантов, который срав-

нивает текстовые фрагменты. А, может быть, вы хотели бы добавить к вашим операторам управления средства жесткого аварийного контроля? Не исключено, что вам захочется иметь оператор цикла **DO**, использующий 32-разрядный индекс. Все это в ваших силах.

По мере дальнейшего изложения материала мы будем приводить примеры применения компилирующих слов. Некоторые из них уже есть в вашей системе. Даже если у вас нет намерения создавать свои компилирующие слова, поняв механизм их создания, вы разберетесь и в том, как образуются собственные компилирующие слова Форта. Форт-система написана на Форте, так что все, что может делать она, можете делать и вы!

Прежде чем перейти к примерам, рассмотрим механизм создания компилирующих слов. Как уже отмечалось, компилирующие слова, когда до них доходит очередь компилятора внутри определения через двоеточие, не компилируются, а выполняются. Это ключ к механизму создания компилирующих слов. Чтобы понять, как он действует, изучим компилятор двоеточия.

Компилятор двоеточия функционирует аналогично текстовому интерпретатору. Он выбирает из входного потока слова и пытается отыскать их в словаре. Однако, вместо того чтобы (как ИНТЕРПРЕТАТОР) исполнять эти слова немедленно, он, как правило, компилирует их адреса в словарь. Но компилятор распознает компилирующие слова и только их исполняет сразу, подобно текстовому интерпретатору.

Каким образом компилятор двоеточия отличает компилирующие слова? По биту немедленного исполнения данного определения (гл. 9 «Структура словарной статьи»): если бит сброшен, то компилируется адрес слова, если установлен, что слово немедленно исполняется. Такие слова называются *словами немедленного исполнения* (immediate).

Слово **IMMEDIATE** делает слово немедленно исполняемым. Его формат:

```
: имя  определение ; IMMEDIATE
```

т. е. это слово выполняется сразу после компиляции определения. Допустим, у нас есть определение:

```
: ТЕСТ   ; IMMEDIATE
```

Это слово немедленного исполнения которое ничего не выполняет. Если мы обратимся к нему из определения другого слова, например:

```
: 2CRS   CR  ТЕСТ  CR ;
```

то будет скомпилирован следующий фрагмент словаря:

2CRS
Поле связи
Поле кода
Адрес CR
Адрес CR
Адрес EXIT

Как видите, определение скомпилировано без слова ТЕСТ. На самом деле оно выполнено во время компиляции слова 2CRS. Поскольку слово ТЕСТ ничего не выполняет, оно бесполезно. Приведем другой пример. Предположим, что у нас есть слово с именем ТЮЛЬПАН и определение:

```
: ТЕСТ   COMPILE ТЮЛЬПАН ; IMMEDIATE
```

Теперь переопределим слово 2CRS точно так же, как и ранее:

```
: 2CRS   CR   ТЕСТ   CR ;
```

и получим следующий результат:

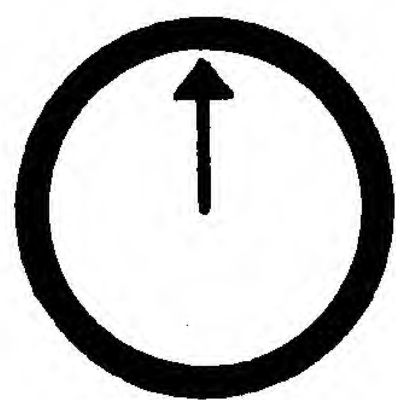
2CRS
Поле связи
Поле кода
Адрес CR
Адрес ТЮЛЬПАН
Адрес CR
Адрес EXIT

На сей раз слово ТЕСТ во время компиляции определения 2CRS скомпилировало адрес слова ТЮЛЬПАН. На самом деле мы нашим определением как бы сказали:

```
: 2CRS   CR   ТЮЛЬПАН   CR ;
```

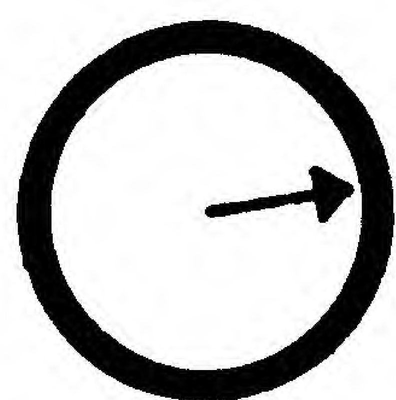
и что компилировать, а что нет, определяет ТЕСТ, потому что это слово немедленного исполнения.

Обратите внимание на слово **COMPILE** (КОМПИЛЯЦИЯ). Мы ввели его как бы между прочим, поскольку его функции проще понять в контексте.



```
: ТЕСТ   COMPILE ТЮЛЬПАН ; IMMEDIATE
```

COMPILE вычисляет адрес следующего слова определения и запоминает его в виде числа:



```
: 2CRS   CR  ТЕСТ  CR ;
```

Когда исполняется слово немедленного выполнения, в котором появилось **COMPILE**, оно компилирует запомненный адрес в создаваемое определение. Этот процесс можно трактовать как отсроченную компиляцию.

Приведем теперь очень полезный пример. Допустим, вы написали отладочное средство с именем ОТЛАД, которое хотите использовать в любом месте своей программы. Так как желательно иметь возможность при необходимости включать и отключать это средство, воспользуемся словом ТЕСТ, определенным следующим образом:

```
: ТЕСТ   ПРОВЕРКА? IF ОТЛАДКА THEN ;
```

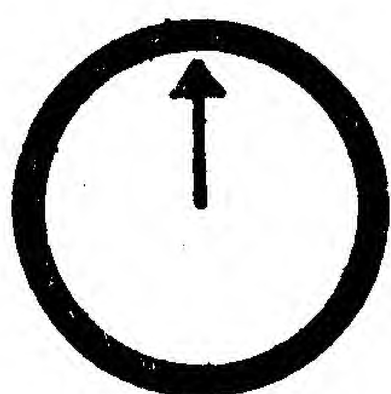
Слово ТЕСТ при выполнении проверяет флаг и определяет, обращаться к слову ОТЛАДКА или нет. Казалось бы, все хорошо, однако много времени уходит на остановку для проверки флага на каждом шаге цикла. Если вы не работаете в отладочном режиме, вам вряд ли нужно всякий раз проверять режим. Проблема решается путем переопределения слова ТЕСТ:

```
: ТЕСТ   ПРОВЕРКА? IF  COMPILE ОТЛАДКА  THEN ;   IMMEDIATE
```

В такой ситуации для включения отладочного средства придется перекомпилировать программу, но это займет немного вре-

мени. В отладочном режиме слово ТЕСТ скомпилирует ОТЛАДКА в соответствующие места программы. В противном случае оно вообще ничего компилировать не будет. Проверка **IF** будет осуществляться в период компиляции.

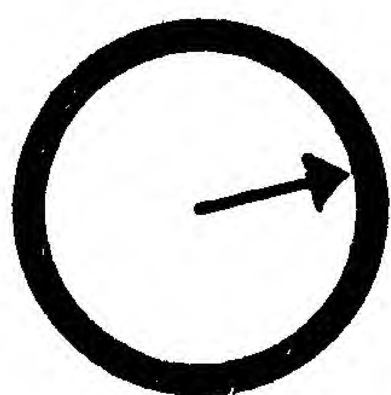
Рассмотрим более сложное, но уже знакомое вам компилирующее слово .", которое не выводит строку на экран, как вы, возможно, склонны думать. На самом деле это слово компилирует строку в словарь с тем, чтобы выдать ее позднее. А какое слово нашу строку затем выводит? Примитив, названный в одних системах **(.")**, в других — **dot"**. Проследим шаг за шагом выполнение данного слова точно так же, как мы это делали применительно к определяющим словам. Приведенное ниже определение слова .", имеется в большинстве Форт-систем, но нам интересен принцип, а не детали.



Фаза 1

```
: dot"   R>  COUNT  2DUP + >R  TYPE ;
: ."   COMPILE dot"  ASCII " STRING ; IMMEDIATE
```

Определение **dot"** и .".



Фаза 2

```
: ВСТРЕЧА   ." Эй, ты " ;
```

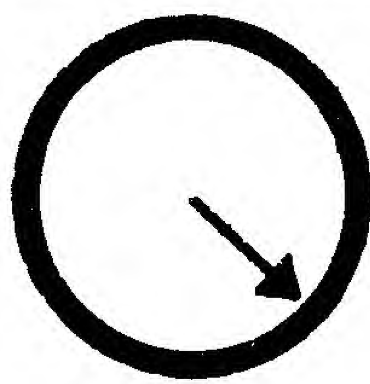
Исполнение .", которое является словом немедленного выполнения (и поэтому исполняется во время компиляции слова ВСТРЕЧА). Оно в свою очередь осуществляет компиляцию:

- адреса слова **dot"** в определение слова ВСТРЕЧА:

ВСТРЕЧА	Поле связи	Поле кода	dot"	...

- строки, ограниченной двойной кавычкой как строки со счетчиком:

ВСТРЕЧА	Поле связи	Поле кода	dot"	7	Э	Й	,	Т	Ы		...



Фаза 3

ВСТРЕЧА

Выполнение слова ВСТРЕЧА, которое вызывает слово **dot"**, а оно уже выводит строку на экран[1].

Следующие два слова применяются при создании новых компилирующих слов:

IMMEDIATE (--) Последнее определенное слово становится немедленно исполняемым, то есть во время компиляции оно будет не компилироваться, а выполняться.

COMPILE xxx (--) Применяется при определении компилирующего слова. Когда это компилирующее слово будет в свою очередь использоваться в исходном определении, адрес поля кода xxx будет скомпилирован в словарную статью, так что когда вновь созданное определение выполняется, выполняется и xxx.

НЕСКОЛЬКО ДОПОЛНИТЕЛЬНЫХ СЛОВ УПРАВЛЕНИЯ КОМПИЛЯЦИЙ

Как вы помните, число, которое появляется в определении через двоеточие, называется литералом (самоопределенным). Например, число 4 в следующем определении является литералом:

```
: ПЛЮС-ЧЕТЫРЕ   4 + ;
```

11	П
Л	Ю
Поле связи	
Поле кода	
(LITERAL)	
4	
+	
EXIT	

[1] *Для очень любознательных.* Слово **dot"** начинает свое выполнение с того, что получает адрес стека возвратов (в нашем случае — адрес строки со счетчиком). **COUNT** преобразует этот адрес отдельно в адрес и значение счетчика для **TYPE**. Но мы должны привести в порядок указатель стека возвратов, чтобы он при возврате показывал бы за строку. Адрес мы вычисляем путем сложения копий адреса и счетчика, полученных с помощью **2DUP**, а затем подкорректированный адрес засылаем в стек возвратов. (Если у вас есть листинги исходных текстов, то, прежде чем экспериментировать, проверьте определение слова .".)

Использование литерала в определении через двоеточие требует двух ячеек. В первой содержится адрес некоторой программы, которая, отработав, поместит содержимое второй ячейки (само число) в вершину стека[1]. Имя этой процедуры может меняться. Назовем ее *кодом периода выполнения для литерала*, или просто **(LITERAL)**. Компилятор, встречая некоторое число, сначала компилирует код периода выполнения для литерала, а затем само число.

Слово **LITERAL** (ЛИТЕРАЛ) вы наиболее часто будете употреблять при компиляции литерала. Это слово компилирует как код периода выполнения, так и само значение, например[2]:

```
4
: ПЛЮС-ЧЕТЫРЕ    ( n -- n+4)   LITERAL + ;
```

Здесь слово **LITERAL** занесет число 4, помещенное в вершину стека перед компиляцией, в элемент словаря как литерал. В результате мы получим элемент словаря, идентичный показанному на приведенном выше рисунке.

Можно привести пример более интересного применения слова **LITERAL**. Вспомните, что в гл. 8 был образован массив с именем ПРЕДЕЛЫ, состоящий из пяти ячеек, в которых хранятся значения предельной температуры для соответствующих горелок. Чтобы упростить доступ к массиву, мы создали слово с именем ПРЕДЕЛ. Эти два определения выглядели следующим образом:

```
CREATE ПРЕДЕЛЫ  10 ALLOT  \  массив из 5 ячеек, содерж.предел.знач.
: ПРЕДЕЛ  ( #горелки -- адрес-пред-знач)  2* ПРЕДЕЛЫ + ;
```

---

[1] *Для борющихся за экономию памяти.* В то время как изображение литерала требует двух ячеек, ссылка на константу занимает только одну ячейку. Таким образом, посредством определения чисел как констант можно экономить память в тех случаях, когда вы используете это число в программе достаточное количество раз, чтобы получаемая экономия перекрыла расходы на создание заголовка константы. Разницу во время выполнения константы и литерала заметить трудно.

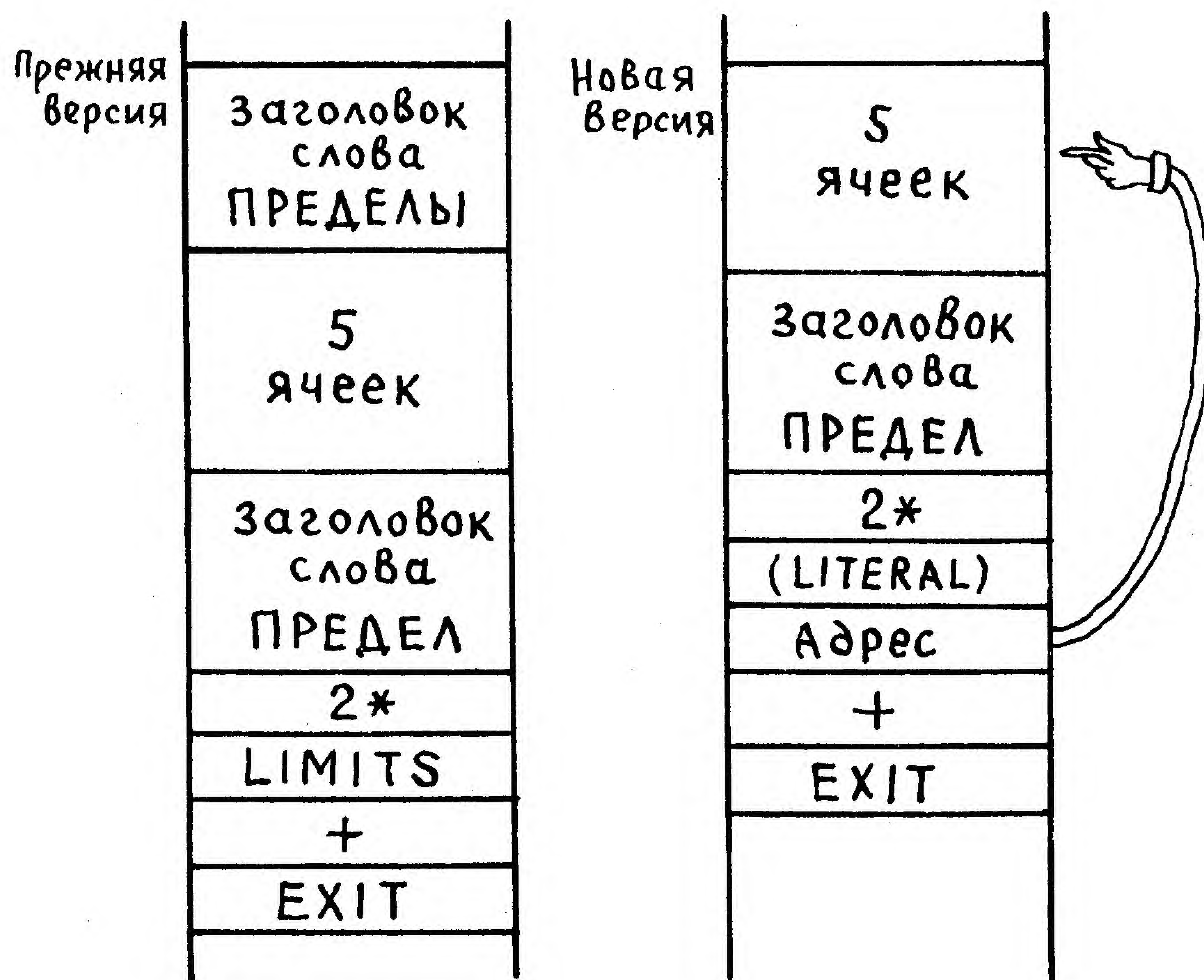
[2] *Для пользователей систем, не допускающих таких действий.* В некоторых Форт-системах при обработке двоеточия запоминается указатель стека данных, а при обработке точки с запятой проверяется соответствие текущего указателя стека и запомненного. Смысл этой проверки заключается в том, чтобы программист не допускал грубых ошибок. После компиляции определения указатель стека должен совпадать с указателем до компиляции. К сожалению, подобный механизм препятствует передаче аргументов слову **LITERAL**. Если ваша система при попытке воспользоваться описанным приемом аварийно завершит работу, «обманите» ее следующим образом:

```
: ИЗВНЕ   ( литерал -- мусор)   0 SWAP ; IMMEDIATE
4
: ПЛЮС-ЧЕТЫРЕ    ИЗВНЕ LITERAL + ; DROP
```

Допустим теперь, что доступ к массиву осуществляется только через слово ПРЕДЕЛ. Мы сможем уничтожить заголовок нашего массива (восемь байтов), сделав такую замену:

```
HERE  10 ALLOT  \ массив из 5 предельных значений
: ПРЕДЕЛ  ( #горелки -- адр-пред-знач)  2* LITERAL + ;
```

В первой строке мы помещаем в вершину стека адрес начала массива (**HERE**), а во второй — заносим этот адрес как литерал в определение слова ПРЕДЕЛ. Таким образом, мы ликвидировали заголовок слова ПРЕДЕЛЫ и сэкономили память словаря.



Существуют еще два слова управления компиляцией, которые вы должны знать, — [и]. Они могут использоваться внутри определения через двоеточие соответственно для прекращения компиляции и ее возобновления. Любые слова, появляющиеся между ними, будут исполнены немедленно, т. е. во время компиляции.

Представьте себе, например, что в некотором определении вы должны вывести строку 3 из блока 180. Чтобы получить адрес третьей строки, вы могли бы воспользоваться выражением

```
180 BLOCK  3 64 *  +
```

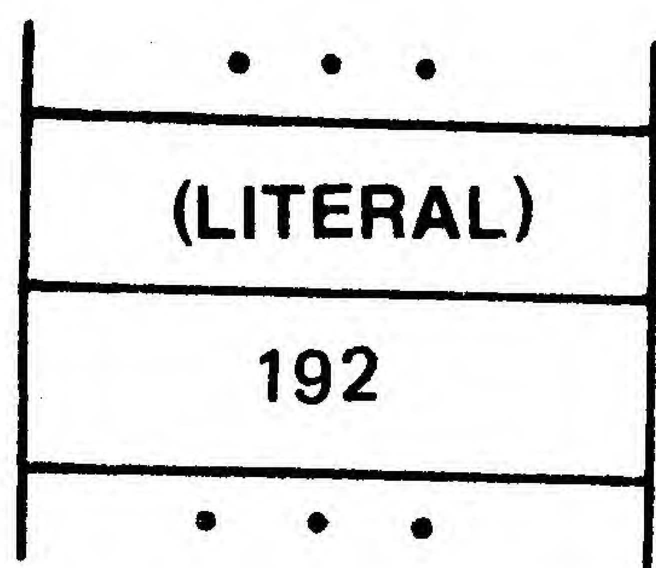
но слишком много времени будет занимать всякий раз при использовании этого определения выполнение фразы: 3 64 *. В качестве альтернативы можно записать следующее:

```
180 BLOCK  192 +
```

однако трудно сразу сообразить, что означает здесь 192. Лучшим решением является такое выражение:

```
180 BLOCK  [ 3 64 * ] LITERAL  +
```

Арифметические операции выполняются только один раз, во время компиляции, а результат заносится в словарь как литерал.



Выше упоминалось о том, что слово] повторно запускает процесс компиляции. На самом деле оно инициируется словом : и во многих системах является компилятором.

Приведем простой пример на применение литерала. Это определение может быть загружено с блока на диске.

```
: НАПЕЧАТАЙ-ЭТО  [ BLK @ ]  LITERAL  LIST ;
```

При исполнении слова ПЕЧАТЬ выводится тот блок, в котором оно определено. (Во время компиляции в **BLK** содержится номер последнего загруженного блока. **LITERAL** заносит этот номер в определение как литерал, так что во время выполнения последней будет служить аргументом для **LIST**.)

Здесь уместно дать определение **LITERAL**:

```
: LITERAL  ( n -- )  COMPILE (LITERAL)  , ; IMMEDIATE
```

Сначала оно компилирует адрес кода периода выполнения, затем — само выражение (используя запятую).

Следующее слово для управления компиляцией — **[COMPILE]**. Допустим, вы хотите переименовать слово **IF**, но делать это так, как показано ниже:

```
: если  IF ; IMMEDIATE
```

не имеете права, поскольку слово **IF** само является немедленно исполняемым. Его код осуществляет переход, если условие не выполняется, на соответствующий оператор **THEN**. Вы должны каким-то образом обойти это препятствие (бит немедленного исполнения) и заставить **IF** компилироваться, как если бы оно было

обычным словом. В такой ситуации вам поможет слово **[COMPILE]**. Если определить

```
: если   [COMPILE] IF ; IMMEDIATE
: иначе  [COMPILE] ELSE ; IMMEDIATE
: то   [COMPILE] THEN ; IMMEDIATE
```

то у вас появится возможность по-новому записывать условия:

```
: ветвление  ( ?)  если  ." Истина "  иначе  ." Ложь "
   то  ." флаг" ;
```

Вас может удивить, почему нельзя использовать слово **COMPILE** непосредственно:

```
: если   COMPILE IF ; IMMEDIATE
```

Вспомните, что **COMPILE** осуществляет так называемую отсроченную компиляцию, т. е. компилирует **IF** не в то слово, которое мы имеем в виду, а в то, которое инициирует это слово (например, «ветвление»). С другой стороны, слово **[COMPILE]** компилирует слова немедленного выполнения в традиционном смысле, иначе они бы исполнялись. Между прочим квадратные скобки в слове **[COMPILE]** означают, что данное слово «выполняется во время компиляции» — таково еще одно соглашение об именовании в Форте. Вы можете смоделировать слово **[COMPILE]** следующим образом:

```
: если   [ ' IF , ] ; IMMEDIATE
```

(или как-нибудь иначе, что воспринимается вашим диалектом языка). В нашем определении мы используем интерпретатор для нахождения адреса **IF**, а затем компилируем этот адрес компиляции в соответствующее определение. Компилятор не допускает немедленного исполнения слова **IF**.

Теперь наступает для вас час испытаний. Если вы его переживете, можете считать себя специалистом по компилирующим словам. Предположим, у нас имеются слова НЕГАТИВНОЕ и -НЕГАТИВНОЕ, которые изменяют обычное изображение на экране негативным и, наоборот, негативное обычным соответственно для любого правильного текста. Наша цель — создать слово **Н."** для автоматического изменения режима экрана на негативный, вывода строки и возвращения к нормальному режиму.

Существуют два решения этой задачи и оба они представляют интерес. Для начала условимся, что изменение режима должно осуществляться тогда, когда строка *выводится, а не компилиру-*

ется. Первое решение заключается в создании слова с именем «нточка"», аналогично слову **dot**", и слово Н.", которое имитирует.", но вместо слова **dot**" компилирует слово «нточка"»:

```
: dot"   R>  COUNT  2DUP + >R TYPE ;
: ."   COMPILE dot"  ASCII " WORD  C@ 1+ ALLOT ; IMMEDIATE
: нточка"   НЕГАТИВНОЕ  R>  COUNT  2DUP + >R  TYPE  -НЕГАТИВНОЕ ;
: Н."   COMPILE нточка"  ASCII " WORD  C@ 1+ ALLOT ; IMMEDIATE
```

Но это решение далеко не изящно и зависит от реализации. Другой вариант — вызов слова .":

```
: Н."   COMPILE НЕГАТИВНОЕ  [COMPILE] ."
   COMPILE -НЕГАТИВНОЕ ; IMMEDIATE
```

Перед вами определение компилирующего слова. Посмотрим, что оно компилирует. Если использовать его в определении

```
: ТЕСТ   Н." Ура!" ;
```

то компилируется следующий фрагмент:

ТЕСТ	Поле связи	Поле кода	НЕГАТИВНОЕ	dot"	4	У	Р	А	!	-НЕГАТИВНОЕ	EXIT

Наше компилирующее слово выполняет три функции:

- компилирует адрес слова НЕГАТИВНОЕ в ТЕСТ (так что НЕГАТИВНОЕ будет выполняться в период исполнения слова ТЕСТ);
- инициирует слово .", которое в свою очередь компилирует **dot"**, и компилирует фрагмент, набранный в строке;
- компилирует адрес слова -НЕГАТИВНОЕ.

Недостаток этого варианта решения заключается в том, что при каждом инициировании Н." компилируются два дополнительных адреса. Первый вариант более эффективен и поэтому предпочтителен в тех случаях, кода у вас имеется исходный текст системы и множество обращений к слову Н.". Второе решение легче реализовать и оно вполне приемлемо при небольшом числе вызовов Н.".

Если вам трудно сразу «переварить» все вышеизложенное, то будем надеяться, что по мере освоения этих слов в процессе практической работы вы испытаете радость познания. Возможно, другие языки и проще в изучении, но скажите, какой иной язык, кроме Форта, позволит вам расширить компилятор?

Как уже отмечалось, наилучший путь освоения Форта — это изучать исходный текст самой Форт-системы (написанный на Фор-

те). Посмотрите, каким образом рассмотренные компилирующие слова используются в других определениях и как они сами определены.

Ниже приведены все дополнительные слова управления компиляцией, введенные в данном разделе.

LITERAL период-компиляции: (n --) период-вполнения: (-- n)	Используется только внутри определения через двоеточие. Во время компиляции значение из стека компилируется как литерал в определение. Во время выполнения это значение будет помещено на стек.
[(--)	Переключение с режима компиляции на режим интерпретации.
] (--)	Переключение на режим компиляции.
[COMPILE] xxx (--)	При использовании внутри определения через двоеточие вызывает компиляцию слова немедленного исполнения xxx , как если бы оно не было словом немедленного исполнения. xxx будет выполняться при выполнении данного определения.

Полезный прием. Ввод с клавиатуры длинных определений. В некоторых Форт-системах нет возможности вводить с клавиатуры определения, состоящие из нескольких строк, потому что при нажатии клавиши RETURN работа в режиме компиляции прекращается. Выход из такого положения — инициировать компилятор в начале каждой строки:

```
: ?ОБъЕМ    ( длина ширина высота -- )<return>
]   6 >  ROT 22 >  ROT 19 > AND AND<return>
]   IF  ." Подходит "    THEN ;<return>ok
```

ФЛАГ СОСТОЯНИЯ

Введем последний термин, имеющий отношение к процессу компиляции, — *состояние*. В большинстве Форт-систем есть переменная с именем **STATE** (СОСТОЯНИЕ), в которой содержится «истина», если вы работаете в режиме компиляции, и «ложь», если вы работаете в режиме интерпретации. Покажем способ вывода значения переменной **STATE**:

```
: .СОСТОЯНИЕ   STATE ? ; IMMEDIATE
```

Введите:

```
.СОСТОЯНИЕ 0 ok
```

В момент вызова .СОСТОЯНИЕ Форт-система находилась в режиме интерпретации (то, что .СОСТОЯНИЕ является словом немедленного исполнения, не играет роли. Интерпретатор не проверяет бит немедленного исполнения).

Теперь вызовите слово .СОСТОЯНИЕ из нового определения:

```
: ТЕСТ   .СОСТОЯНИЕ ; -1 ok
```

На сей раз значение **STATE** равно «истине», поскольку .СОСТОЯНИЕ было инициировано компилятором.

В каких случаях возникает необходимость знать состояние? Всегда, когда вы хотите создать слово, которое должно *«делать вид»*, что его поведение одинаково как внутри определения, так и вне его, а на самом деле оно проявляет себя по-разному. В качестве примера можно привести слово **ASCII**. При обычном использовании это слово появляется внутри определения через двоеточие:

```
: ТЕСТ  ( -- ascii-a)  ASCII A ;
```

При *исполнении* слова ТЕСТ в вершину стека вносится число 65. **ASCII** осуществляет преобразование во время компиляции. Оно должно быть компилирующим словом: выбирать из входного потока символ, компилировать его как литерал, чтобы последний мог быть занесен в стек во время выполнения слова ТЕСТ. Вы можете создать компилирующий вариант слова **ASCII** следующим образом:

```
: ASCII  ( --c )
\ Компиляция:   c    ( -- )
   BL WORD 1+ C@  [COMPILE] LITERAL ; IMMEDIATE
```

(*Примечание.* Стековый комментарий в первой строке определяет поведение слова во время выполнения — это синтаксис использования слова. Комментарий во второй строке показывает, что должно произойти во время компиляции, в частности должно быть выполнено считывание символа из входного потока и ничего не ставлено в стеке.)

В приведенном выше определении слово **WORD** выбирает из входного потока текст, ограниченный пробелом, и вносит в вершину стека адрес участка памяти, где будет храниться строка со счетчиком. Мы считаем, что такой текст содержит только один символ, поэтому пропускаем счетчик байтов посредством 1+ и выбираем значение с помощью **C @** . Затем инициируем слово **LITERAL**, компилирующее код периода выполнения для литерала, за которым следует само значение. Это все; что нам требуется.

Попытаемся заставить **ASCII** выполняться *вне определения*. Например, выражение

```
ASCII A
```

внесет в вершину стека значение 65 (что может оказаться полезным при составлении таблиц и т. д.). Далее «попросим» **ASCII** выполнить что-нибудь такое, что оно еще не делало. Создадим следующий вариант определения:

```
: ASCII  ( -- c)  BL WORD 1+ C@ ;
```

Для того чтобы одно и то же слово **ASCLL** могло выполняться в обоих вариантах, оно должно реагировать на состояние

```
: ASCII  ( -- c)
\ Компиляция:   c   ( -- )
\ Интерпретация:   c   ( -- c)
   BL WORD 1+ C@
   STATE @ IF  [COMPILE] LITERAL  THEN ; IMMEDIATE
```

Такое слово называется *зависимым от состояния.* На первый взгляд зависимость от состояния кажется свойством разумным и даже желательным, особенно для слова **ASCII**, которое используется только при компиляции. Однако когда программист пытается повторно применять зависимые слова в другом определении и хочет, чтобы они выполнялись обычным образом, возникают трудности. Приведем пример использования компилирующего слова, зависимого от состояния, в определении другого компилирующего слова. По мере того как растет глубина вложенности слова, зависимого от состояния, программисту приходится затрачивать все больше усилий на то, чтобы держать под контролем, что и где должно выполняться.

Применение простых слов безопаснее, чем зависимых от состояния, так как их поведение предсказуемо. В некоторых ранее созданных системах слово **."** зависело от состояния. С введением Стандарта-83 оно было разделено на два отдельных слова, управляющих использованием во время компиляции (**."**) и в период выполнения во время интерпретации блоков при их загрузке (**.(**).

Слово **'** постигла участь описанных выше слов и теперь это простое слово. Его функции внутри определения аналогичны функциям, выполняемым во время интерпретации: оно выявляет определение, имя которого находится во входном потоке во время исполнения. Вариант апострофа, функционирующего как компилирующее слово, называется **[']**.

Некоторые разработчики Форта считают, что зависимость от состояния не имеет права на существование. Применительно к **ASCII** возможны два решения: 1) разрешить его использование только внутри определения и, если возникает необходимость, создать слово с другими функциями, скажем **ascii** (на нижнем регистре), для режима интерпретации; 2) сделать его пригодным только для интерпретации:

```
: TECT   [ ASCII A ] LITERAL ;
```

Можно договориться и ввести другое слово:

```
: [ASCII]   ASCII  [COMPILE] LITERAL ;  IMMEDIATE
```

которое рекомендуется применять следующим образом:

```
: TECT   [ASCII] A ;
```

ВВЕДЕНИЕ В БЛОК-СХЕМЫ ФОРТА

Применение блок-схем обеспечивает наглядность логической структуры определений. На блок-схеме хорошо видно, в какое место осуществляются переходы и какие фрагменты выполняются циклически. Традиционные блок-схемы не вполне подходят для структурной организации Форт-программ, поэтому программисты предпочитают им иные схемы. Вопрос о том, какие схемы являются более подходящими для программирования на Форте, остается открытым. Каждый программист решает его для себя сам. Тема эта довольно обширна и выходит за рамки данной книги.

Рассматриваемые здесь диаграммы основаны на так называемых D-схемах. Последовательные операторы записывают один под другим, не соединяя их линиями и не заключая в рамки:

оператор
следующий оператор
следующий оператор

Линии же служат для того, чтобы показать, что действия выполняются не в порядке очередности, а либо в зависимости от некоторого условия (условные операторы), либо неоднократно (операторы цикла). Условный оператор Форта

```
условие IF истина ELSE ложь THEN оператор
```

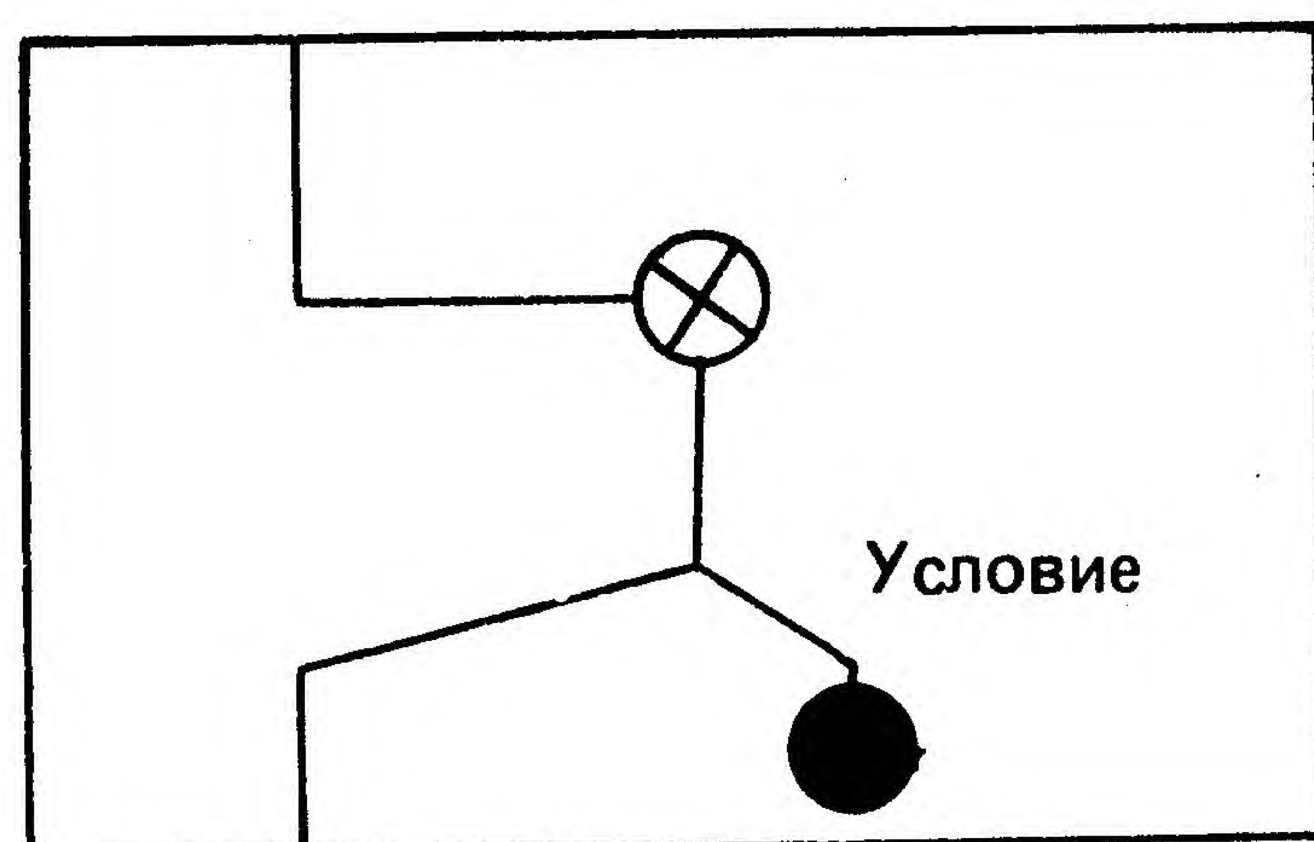
изображается следующей диаграммой:



Если выражение в какой-либо альтернативной ветви опущено, то в этом месте проводится сплошная вертикальная линия: Расположение истинной и ложной ветвей (слева и справа) не имеет значения.

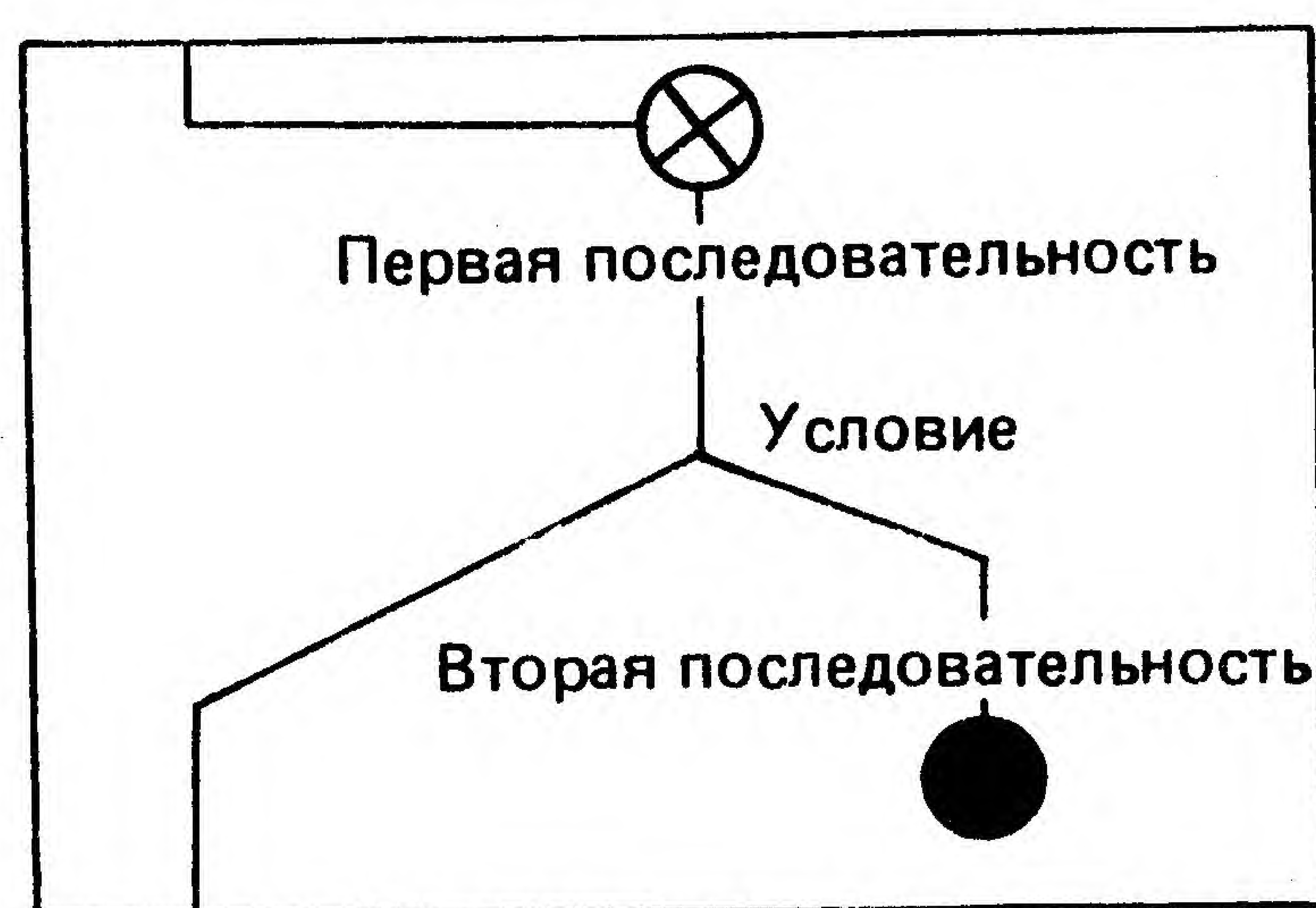


Структура **BEGIN ... UNTIL** изображается так:



Циклическая структура располагается справа от основной линии вычислений и соединяется с ней горизонтальной линией в верхней своей части. Если нужно показать вложенные циклы, то они должны быть расположены правее. Черная точка является признаком конца цикла. Она означает, что управление возвращается в точку начала, обозначенную символом X, взятым в кружок. В зависимости от условия этот цикл либо будет повторен, либо завершится. Диагональная линия влево вниз отображает возврат на внешний уровень выполнения.

Цикл вида **BEGIN ... WHILE ... REPEAT** имеет аналогичную диаграмму:



Итак, мы кратко осветили вопрос о применении блок-схем при программировании на Форте и теперь можем наглядно представить вам структуру двух очень важных слов.

ЗАКЛЮЧЕНИЕ

Мы завершаем рассмотрение текстового интерпретатора и компилятора и в конце этого раздела, возможно, увидим их несколько в ином свете.

В процессе изложения мы неоднократно упоминали слово

INTERPRET, имея в виду текстовый интерпретатор. Его строгое описание выглядит так:

```
INTERPRET    ( -- )   Интерпретация текста из входного потока по ука-
                      зателю >IN до исчерпания входного потока.
```

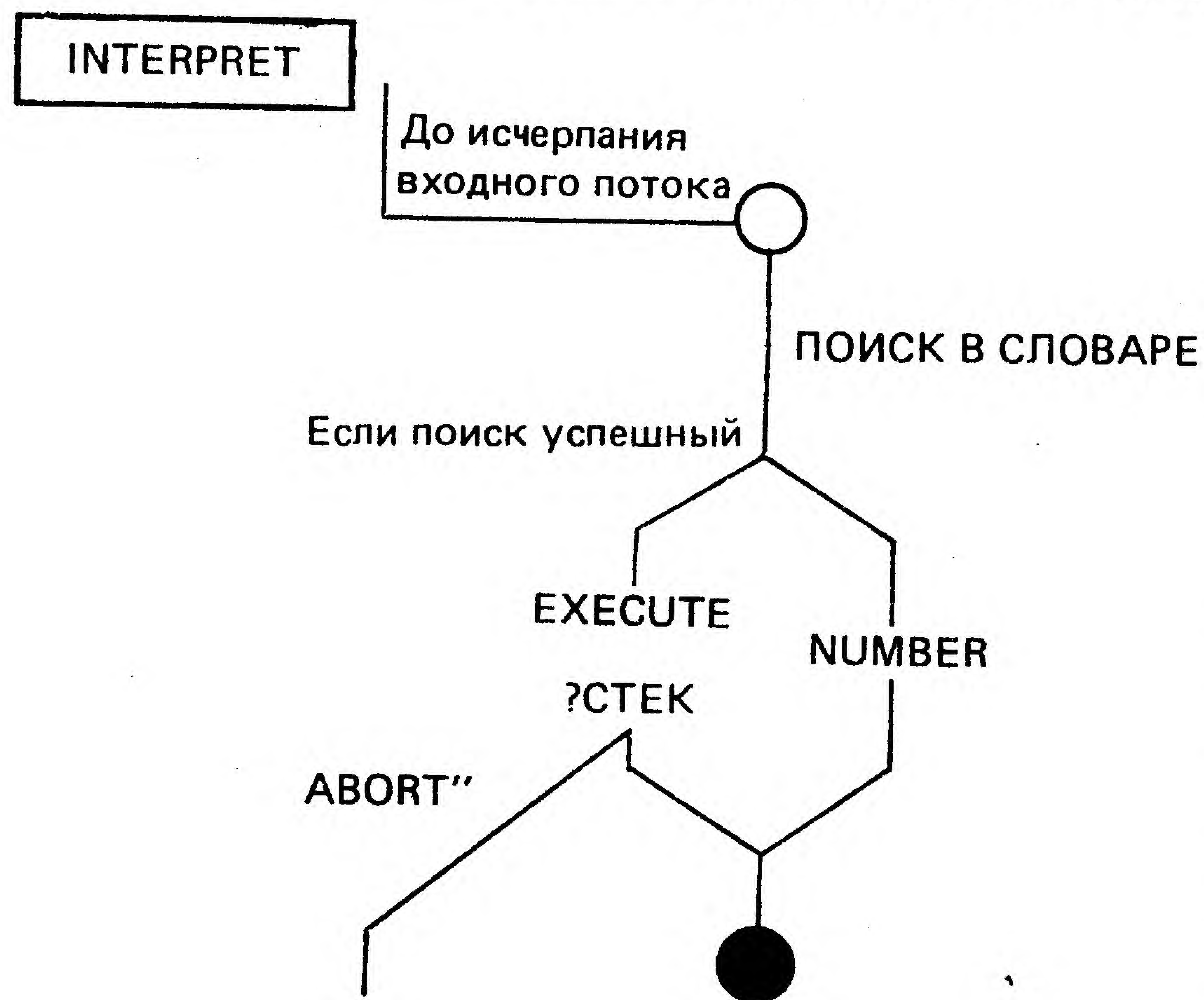
Несмотря на то что это слово первоначально предназначалось для использования самой Форт-системой, оно может применяться и в ваших программах. Предположим, вы написали цикл, а он выполняется не так, как вам нужно. Для отладки в этой ситуации пригодилось бы слово, которое останавливало бы выполнение программы на каждом шаге цикла и позволяло бы ввести ряд команд в диалоговом режиме, причем после нажатия клавиши RETURN выполнение цикла должно продолжаться. Подобное отладочное средство можно организовать с помощью **INTERPRET**.

Создадим следующее определение:

```
: ТЕСТ   0 BEGIN  DUP .  1+  QUERY INTERPRET  0 UNTIL ;
```

Введите слово ТЕСТ. Оно выведет нуль, остановится и будет ждать. Если вы нажмете клавишу RETURN, цикл продолжит свое выполнение, и на экране высветится единица. До нажатия клавиши RETURN вы можете ввести любую команду. Прежде чем продолжить выполнение цикла, **INTERPRET** ее выполнит. Если вы хотите завершить цикл, введите **QUIT** или сделайте ошибку, вызывающую **ABORT**. Любое из этих действий очистит стек возвратов и тем самым приведет к выходу как из **INTERPRET**, так и из ТЕСТ. (Это средство можно использовать гораздо шире [1].)

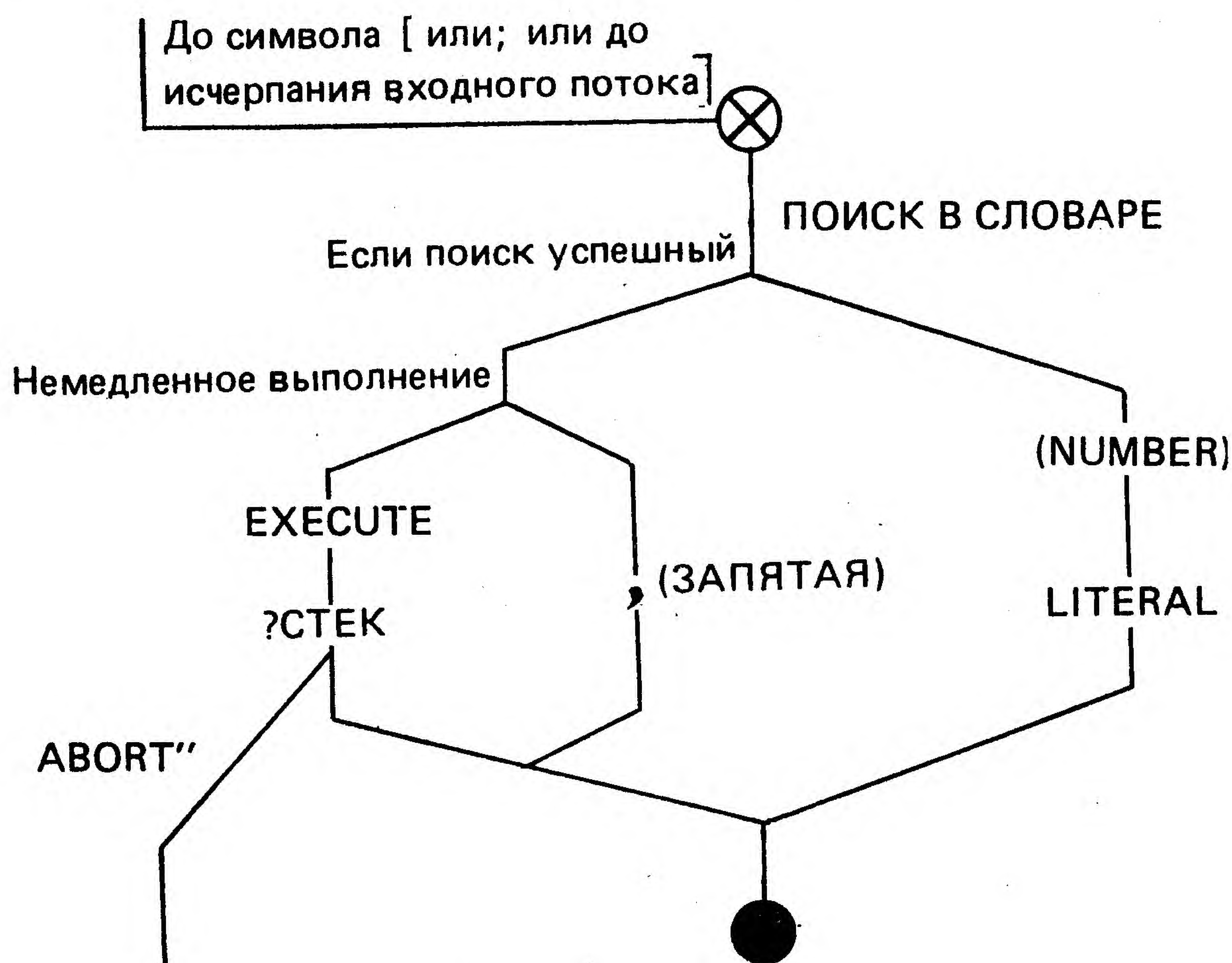
В разных диалектах Форта слово **INTERPRET** определено поразному, но суть этого слова можно передать, изобразив алгоритм его выполнения с помощью D-схемы.



Алгоритм выполнения слова **INTERPRET** можно описать следующим образом. Начинаем цикл. В теле цикла выбираем очередное слово из входного потока и осуществляем поиск его определения в словаре. Если определение найдено, исполняем слово. Затем проверяем, не исчерпан ли стек.

(В том случае, когда стек исчерпан, завершаем цикл посредством **EXIT** и выдаем аварийное сообщение.) Если слово в словаре не найдено, пытаемся преобразовать введенный фрагмент в число и внести его значение в вершину стека. Далее повторяем цикл входного потока.

Для сравнения опишем алгоритм компилятора двоеточия. Начинаем цикл. В теле цикла выбираем очередное слово из входного потока и осуществляем поиск его определения в словаре. Если определение найдено, — это слово Форта. Слово немедленного исполнения сразу выполняем и проверяем, не исчерпан ли стек. Для слова, не имеющего признака немедленного исполнения, определяем адрес компиляции. Если определение слова в словаре не найдено, пытаемся преобразовать его в число и скомпилировать в качестве литерала. Затем повторяем цикл до тех пор, пока не исчерпается входной поток или система не будет переведена в режим интерпретации. Схема рассмотренного алгоритма:



(Если вам доступен исходный текст системы, мы рекомендуем вам изучить настоящие определения слов **INTERPRET** и].)

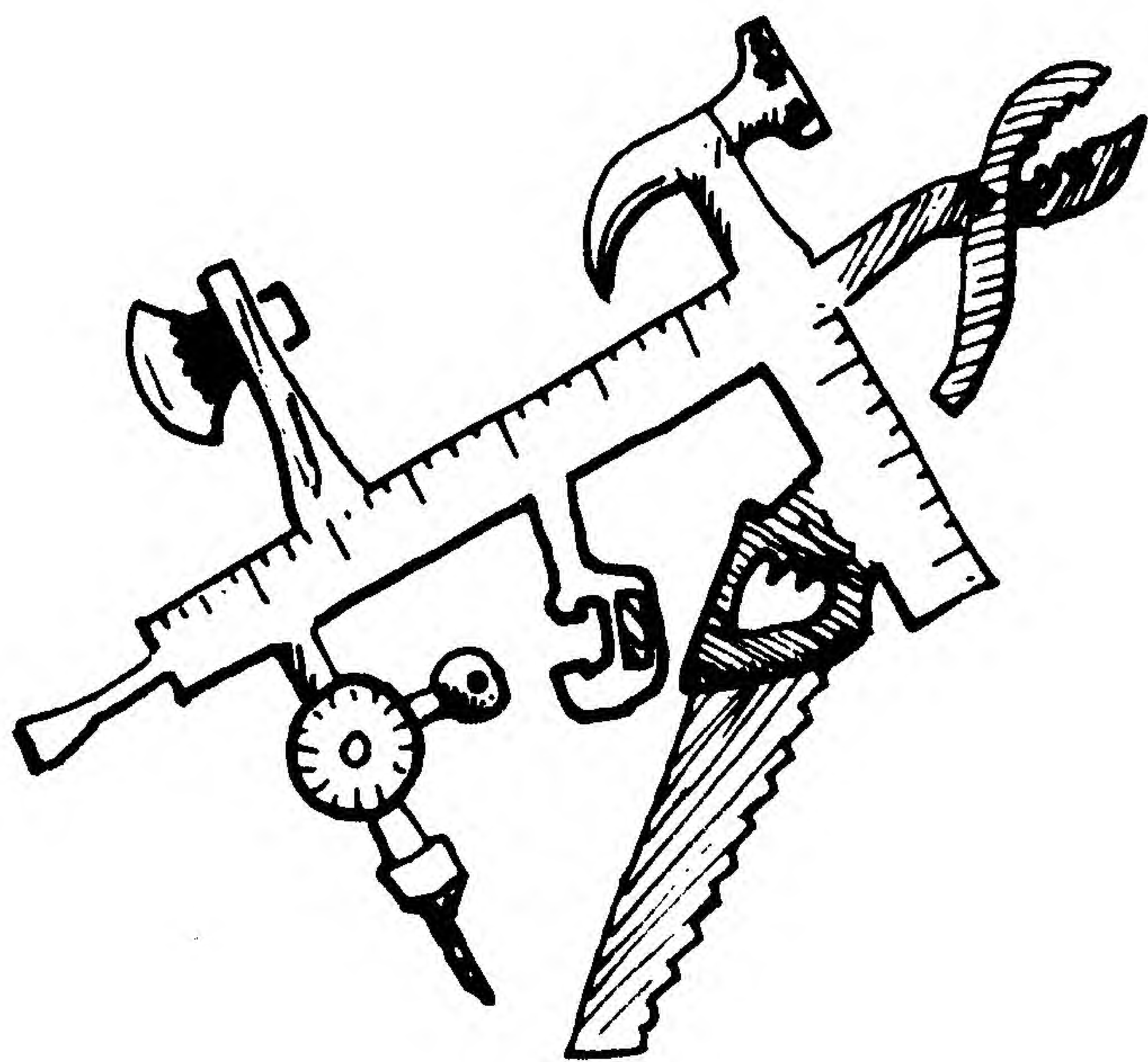
Сравните обе диаграммы, и вы увидите, что интерпретатор может вызывать] для того, чтобы определить, будет данное слово выполняться или компилироваться. Такая простота в организации системы позволит вам легко добавлять новые компилирующие слова.

Итак, существуют два способа расширения Форт-компилятора:

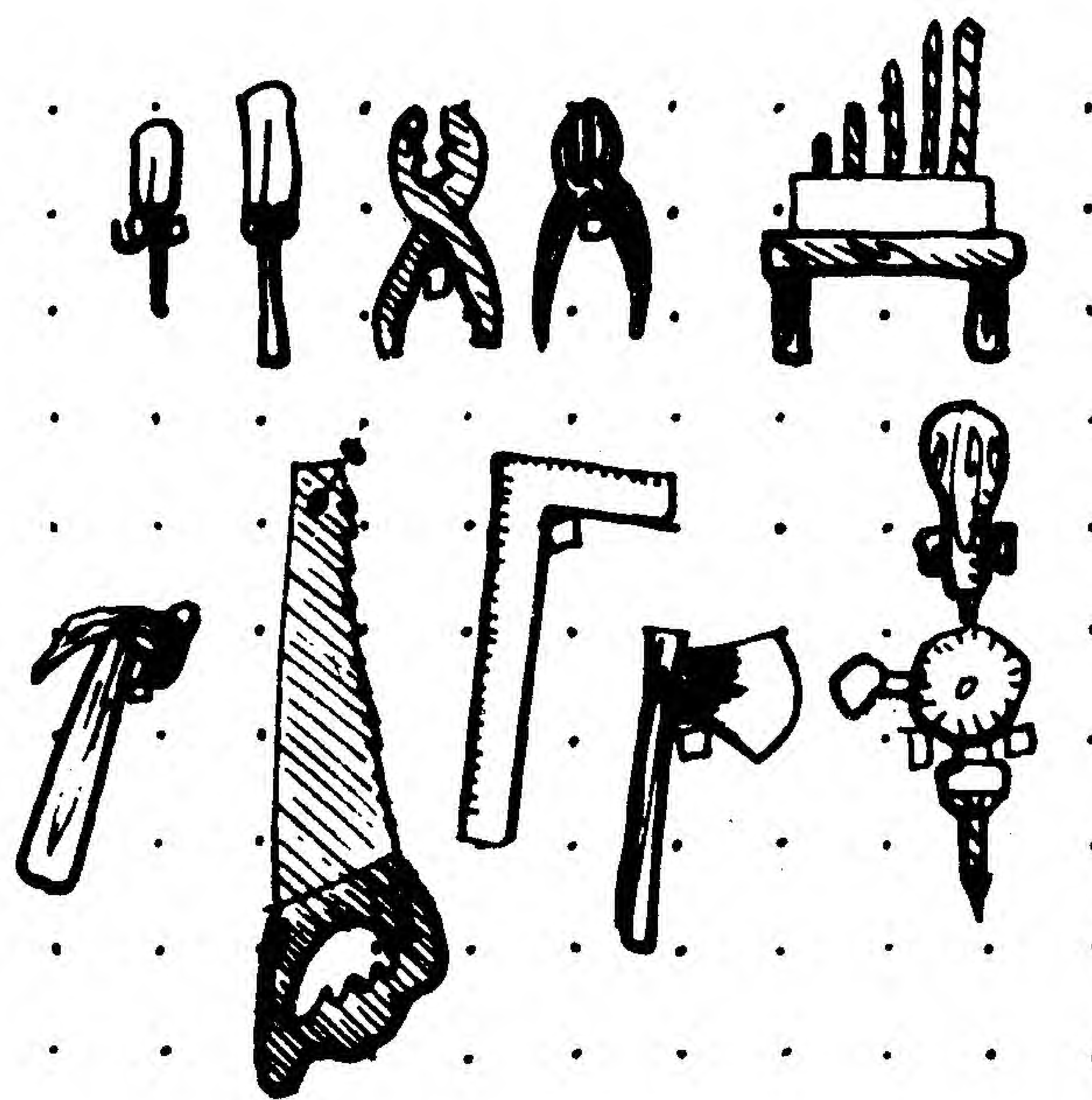
- добавление новых, специализированных компиляторов посредством создания новых определяющих слов;
- расширение имеющегося компилятора посредством создания новых компилирующих слов.

В то время как традиционные компиляторы претендуют на роль универсальных инструментальных средств, Форт-компилятор представляет собой набор отдельных простых инструментов и позволяет на их основе создавать дополнительный инструментарий. Что, на ваш взгляд, лучше:

СЛОЖНОСТЬ



ИЛИ ПРОСТОТА?



Ниже приводится перечень слов Форта, рассмотренных в настоящей главе.

DOES>	период-выполнения: (-- a)	Используется при создании определяющих слов. Отмечается конец участка периода компиляции и начала участка периода выполнения. Операции периода выполнения определены на высокоуровневом Форте. Во время выполнения на стеке должен находиться pfa определенного слова.
IMMIDIATE	(--)	Последнее определенное слово становится немедленно исполняемым, то есть во время компиляции оно будет не компилироваться, а выполняться.
COMPILE xxx	(--)	Применяется при определении компилирующего слова. Когда это компилирующее слово будет в свою очередь использоваться в исходном определении, адрес поля кода xxx будет скомпилирован в словарную статью, так что когда вновь созданное определение выполняется, выполняется и xxx.
LITERAL	период-компиляции: (n --) период-вполнения: (-- n)	Используется только внутри определения через двоеточие. Во время компиляции значение из стека компилируется как литерал в определение. Во время выполнения это значение будет помещено на стек.
[	(--)	Переключение с режима компиляции на режим интерпретации.
]	(--)	Переключение на режим компиляции.

[COMPILE] xxx	(--)	При использовании внутри определения через двоеточие компиляция слова немедленного исполнения xxx , как если бы оно не было словом немедленного исполнения. xxx будет выполняться при выполнении данного определения.
STATE	(-- a)	Переменная, содержащая флаг состояния системы: истину при компиляции и ложь при интерпретации.
INTERPRET	(--)	Интерпретация текста из входного потока по указателю >IN до исчерпания входного потока.

ОСНОВНЫЕ ТЕРМИНЫ

Бит использования во время компиляции. Бит элемента словаря, который определяет, будет ли слово во время компиляции выполняться, а не компилироваться.

D-схема. Графическое представление логической структуры некоторой программы, а в случае Форта — некоторого определения.

Компилирующее слово. Слово, используемое внутри определения через двоеточие для выполнения действий во время компиляции.

Немедленно исполняемое слово. Словарная статья, где установлен бит непосредственного использования, что вызывает во время компиляций не компилирование данного слова, а его выполнение.

Определяющее слово. Слово, которое при своем выполнении создает новый элемент словаря. Определяющее слово задает функции периода-компиляции и периода-выполнения для каждого члена семейства слов, определяемых этим словом.

Функции периода выполнения. По отношению к *определяющим словам*: последовательность команд, которые будут выполняться при выполнении любого слова из некоторого семейства. По отношению к *компилирующим словам*: некоторая программа, которая будет выполняться при выполнении определения-родителя. Не у всех компилирующих слов имеются функции периода выполнения.

Функции периода компиляции. Применительно к *определяющим словам*: последовательность команд, которая будет выполняться при выполнении этого определяющего слова — команды осуществляют компиляцию слов, принадлежащих к конкретному семейству. Применительно к *компилирующим словам*: функции слова, содержащегося в определении через двоеточие, которые выполняются во время компиляции данного определения.

УПРАЖНЕНИЯ

11.1. Введите определяющее слово с именем ЗАГРУЗКА, которое будет определять слова, загружающие при своем исполнении некоторый блок. Например, выражение

```
600 ЗАГРУЗКА КОРРЕСПОНДЕНЦИЯ
```

должно определить слово КОРРЕСПОНДЕНЦИЯ. В результате выполнения последнего должен быть загружен блок 600.

11.2. Определите слово с именем СИСТЕМА., создающее слова для вывода чисел в конкретных системах счисления. Например, выражение

```
16 СИСТЕМА. Н.
```

должно определить слово Н., которое выводило бы значение из вершины стека в шестнадцатиричной системе, но саму систему счисления (**BASE**) не меняло бы:

```
DECIMAL
17 DUP Н. .<return> 11 17 ok
```

11.3. Определите слово с именем МНОГО, чтобы оно получало адрес некоторого слова, например **CR** или STAR, и создавало множественное число этого слова, в нашем случае CRS или STARS. Вы будете задавать адрес слова в единственном числе посредством апострофа. Например, фраза

```
' CR  МНОГО CRS
```

определит слово CRS так же, как и выражение

```
: CRS   ?DUP IF  0 DO CR  LOOP  THEN ;
```

11.4. На французский язык слова **DO** и **LOOP** переводятся как TOURNE и RETOURNE соответственно. Определите TOURNE и RETOURNE как французские имена слов управления циклом, используя слова **DO** и **LOOP**, после чего проверьте их, написав цикл на французском языке.

11.5. Напишите слово с именем РАЗ, которое вызовет исполнение оставшихся во входном потоке символов до возврата каретки. Число исполнений задается числом, находящимся в вершине стека, например:

```
7 РАЗ  42 EMIT  SPACE<return> * * * * * * * ok
```

11.6. В настоящей главе демонстрировался пример использования определяющего слова с именем ФОРМА. Чтобы задать конкретное изображение, нужно было ввести восемь чисел, каждое из которых представляло собой битовый шаблон отдельной строки. Придумайте более изящный способ задания описания рисунка, содержащего 8 $\times$ 8 элементов изображения. Можно, например, вместо шестнадцатиричных цифр задавать некоторую схему.

Теперь реализуйте то, что вы придумали (с привлечением существующего определения слова .РЯД). Проверьте свой способ формирования рисунка.

ЛИТЕРАТУРА

1. "Add a Break Point Tool," *Forth Dimensions*, Vol. V, No. 1, p. 19.

Глава 12
ТРИ С ПОЛОВИНОЙ ПРИМЕРА

Программирование на Форте в большой мере, чем на любом другом языке, является искусством. Форт подобен кисти художника — этот язык предоставляет программисту средства, позволяющие ему полностью контролировать свои действия. По словам Ч. Мура, хороший программист с помощью Форта может выполнить работу блестяще, плохой — загубить дело. Программист должен чувствовать «стиль» Форта. Предлагаем вашему вниманию некоторые элементы хорошего стиля программирования на Форте:

· простота;

· предпочтение большого числа коротких определений небольшому числу длинных;

· соответствие между названиями слов и описываемыми или легко воспринимаемыми действиями или структурами данных;

· правильный выбор имен;

· удачное разбиение программы на блоки и понятный комментарий.

Неплохим методом освоения стиля программирования на Форте, свободным от проб и ошибок, представляется разбор существующих программ, написанных на Форте, в том числе и самой Форт-системы. Мы включили в книгу определения многих слов Форт-системы и рекомендуем вам продолжить ее изучение самостоятельно.

В настоящей главе будут рассмотрены три примера. В первом примере показано использование хорошо выделенных определений и создание специализированного прикладного «языка». Второй пример иллюстрирует способ преобразования математического выражения в определение на Форте. Вы увидите, как с помощью арифметических операций над числами с фиксированной точкой можно увеличить скорость выполнения программ и уменьшить их объем. В третьем примере речь идет о применении конструкций ассемблера Форта и демонстрируется мощь определяющих слов. И

наконец, без всяких пояснений мы предоставим вам возможность научиться разбираться в тексте программ, написанных на Форте.

ОТКАЧКА ФАЙЛА

Итак, наш первый пример — простая файловая система[1]. Это серьезная и полезная программа, которая к тому же является неплохим пособием для изучения хорошего стиля программирования на Форте. Мы разделили этот раздел на три части:

- рекомендации конечному пользователю по работе с файловой системой;
- описание структуры программы и выполнения некоторых определений;
- листинг программы с блоками, содержащими документацию.

Как пользоваться простой файловой системой. Рассматриваемая здесь файловая система позволяет быстро запоминать и восстанавливать информацию. Она мгновенно запоминает (для последующего применения) фамилии людей, их адреса и телефоны[2]. Вы можете не только вводить, изменять и удалять записи, но и находить файл с любой информацией. Например, по номеру телефона легко установить фамилию абонента, по известной фамилии определить место работы и т. д.

Для каждого человека отводится некоторая *запись*, которая состоит из четырех *полей* с именами: фамилия, имя, работа, телефон.

Поиск информации. Вы можете просматривать файл в поисках содержимого какого-либо поля, используя слово НАЙТИ, за которым должны следовать имя поля и его содержимое:

```
найти работа диктор<return> Дан Рэйзер ок
```

Если в поле «работа» содержится строка «диктор», то система выведет фамилию диктора. При отсутствии файла с такими атрибутами система выдаст сообщение: «Сведений нет». В том случае, когда поле с искомыми атрибутами найдено, запись с соответствующей информацией становится *текущей*. Вы можете вывести содержимое любого поля текущей записи с помощью слова «дать». Например, если вы ввели упомянутую выше строку, то теперь можете написать:

```
дать телефон<return> 555-9876 ок
```

---

[1] *Для пользователей файловой системы.* Версии Форта, поставляемые профессиональным программистам, включают намного больше средств для работы с базами данных.

[2] *Для программистов.* Вы легко можете изменить имена или увеличить число полей, обрабатываемых системой.

Команда «найти» применяется только для поиска *первого* поля с указанными атрибутами. Для того чтобы добраться до следующего такого поля, воспользуйтесь командой «еще». В частности, чтобы найти еще одного диктора, нужно ввести

```
еще<return> Конни Чанг ok
```

а затем

```
еще<return> Франк Рейнольдз ok
```

Если в данном файле больше нет сведений о дикторах, то при посылке команды «еще» вы получите сообщение: «Больше нет», т. е. полей с такими атрибутами в файле не осталось.

Для получения списка лиц, у которых в соответствующем поле информация совпадает с атрибутами, применявшимися при последнем поиске, введите команду «все»:

```
все
Дэн Рэйзер
Конни Чанг
Фрэнк Рейнольдз
 ok
```

Так как фамилия и имя хранятся порознь, вы можете организовать поиск с помощью команды «найти» по одному из этих атрибутов. Но если вам известны и имя, и фамилия человека, которого вы ищете, то для экономии времени можно осуществлять поиск сразу по двум полям, используя слово «фио». С этой целью вы должны задать слову «фио» имя и фамилию, причем фамилию нужно задать первым операндом и отделить его от второго операнда запятой, например:

```
фио Уандер,Стив<return> Стив Уандер ok
```

(после запятой не должно быть пробела, поскольку запятая отмечает конец первого поля и начало второго). Подобно командам «найти» и «еще» слово «фио» выводит найденное имя.

С помощью слова «пара» вы можете осуществить поиск *любой пары* полей, для чего необходимо задать имена обоих полей и их содержимое, разделив операнды запятой. В частности, чтобы найти некоего комментатора по имени Дан, вы должны ввести:

```
пара работа диктор,имя Дан<return> Дан Рэйзер ok
```

Сопровождение файлов. Если вам требуется ввести новую запись, вы должны применить команду «внести», разместив за ней

операнды: фамилия, имя, место работы, телефон, причем операнды отделяются только запятой, например:

```
внести  Нуреев,Рудольф,танцовщик балета,555-1234<return>_ok
```

Изменить содержимое единственного поля внутри текущей записи вы можете с помощью команды «изменить», расположив за ней имя этого поля, а затем новое содержимое последнего:

```
изменить работа хореограф<return>_ok
```

Для удаления текущей записи вы должны ввести команду «удалить»:

```
удалить<return>_ok
```

После того как вы что-то добавляли к записям, модифицировали их либо удаляли, обязательно введите слово **FLUSH**, если собираетесь отключить компьютер или сменить диск:

```
FLUSH_ok
```

ПРОГРАММИСТУ О СТРУКТУРЕ ПРИКЛАДНОЙ ПРОГРАММЫ

В настоящем разделе мы дадим рекомендации начинающему Форт-программисту по работе с листингом программы, который приводится ниже. Вы познакомитесь со структурой этой программы и некоторыми из наиболее сложных определений.

Итак, обратимся к листингу. Обратите внимание на то, что вид данной программы несколько отличается от всех остальных, показанных в книге. Здесь использованы соглашения, принятые в фирме FORTH, Inc.

Блоки, расположенные на левой стороне разворота, называются *блоками сопровождения*. Они содержат комментарий к исходному тексту блоков на правой стороне разворота. Например, определение слова HELP располагается в строке 1 блока 240, а комментарий к этому слову — в строке 1 блока 561. Наличие блоков сопровождения имеет очевидные преимущества. Прежде всего документация всегда находится вместе с программой. В большинстве систем есть слово, которое обеспечивает связь между исходным текстом блока и комментарием к блоку в обе стороны (в полиФорте **Q**). Программист получает возможность в ходе разработки программы создавать документацию и вносить в нее изменения. С помощью слов **LOCATE** и **Q** можно сделать доступным описание, относящееся к любому слову.

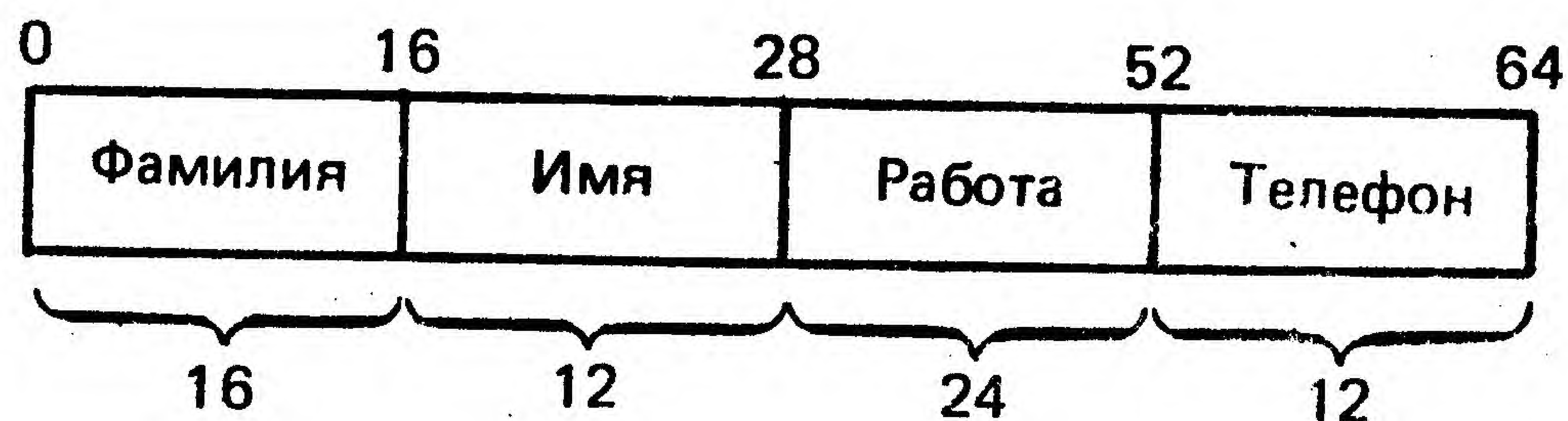
Если группа слов описана в одном блоке, то и комментарий к ним должен размещаться в одном блоке, что спасает от повторных описаний и многословия, свойственного глоссариям, где материал расположен в алфавитном порядке.

Слово HELP определено в блоке 240. Оно выводит текст *блока подсказок* (блок 561), который содержит все команды уровня пользователя. Блок подсказок появляется при загрузке блока 240 (так что он доступен пользователю при компиляции программы) или при вводе пользователем слова HELP

Выражение 241 243 THRU в строке 2 блока 240 загружает фрагмент программы более низкого уровня. Остальные операторы блока загружают слова самого высокого уровня. Помещая группу команд в блок загрузки, а не в последний блок, пользователь тем самым помещает комментарий к этим командам в блок сопровождения блока загрузки. Обратите внимание на девять команд пользователя в блоке 240. Как просты их определения, несмотря на то, что они реализуют очень мощные функции.

Рассматриваемая программа является образцом хорошо выполненной разработки на Форте. Слово —НАЙТИ (примитивное слово для работы с файлами) выделено таким образом, что его можно включать в определения таких слов, как «найти», «еще», «все» внутреннего слова (ПАРА), которое используется словами «пара» и «фио». Мы кратко разберем эти определения, но сначала обсудим общую структуру программы. Одной из основных ее особенностей является то, что каждое из четырех полей имеет имя, которое необходимо ввести, чтобы задать соответствующее поле. Например, выражение «фамилия ПОМЕСТИТЬ» поместит строку символов из входного потока в поле «фамилия» текущей записи, выражение «фамилия .ПОЛЕ» распечатает содержимое данного поля и т. д.

Существует информация двух типов для обозначения полей: начальный адрес поля относительно начала записи и его длина. В рассматриваемой программе структура записи такова:



Например, поле «работа» начинается с 28-го байта каждой записи, а его длина составляет 24 байта. Мы вправе сделать длину записи равной в точности 64 байтам, чтобы при распечатке файла посредством LIST столбцы были бы выровнены по вертикали. Такая структура записи выбрана из соображений удобства при программировании именно нашей задачи. Если вы внесете изменения

в программу, то можете сделать записи нужной вам длины и состоящими из требуемого числа полей[1].

Две величины, характеризующие каждое поле, мы помещаем в таблицу двойной длины, связанную с именем этого поля. Следовательно, наше определение «работа» будет иметь вид:

```
CREATE работа  28 ,  24 ,
```

6	Р
А	Б
Поле связи	
Поле кода	
28	
24	

Таким образом, при вводе имени поля в вершину стека вносится адрес таблицы, описывающей поле «работа», по которому мы можем выбрать любую из двух характеризующих его величин. Назовем каждый такой элемент *таблицей спецификации поля*, или для краткости *таблицей полей.*

Часть функций нашей программы обусловлена требованиями, предъявляемыми командами «найти», «еще» и «все», т. е. по коман-

---

[1] *Для тех, кто хочет модифицировать нашу файловую систему.* При изменении параметров полей убедитесь, что начальный байт каждого последующего поля правильно стыкуется с предыдущим полем. Например, если первое поле имеет длину 30 байт после такого выражения

```
CREATE 1ПОЛЕ   0 ,  30 ,
```

то начальный адрес следующего поля должен быть 30-м и определяться выражением

```
CREATE 2ПОЛЕ  30 ,  12 ,
```

и т. д.

Установите значение /ЗАПИСЬ равным длине всей записи (начальный байт последнего поля плюс длина поля). Используя /ЗАПИСЬ, система автоматически подсчитывает число записей, которые она может разместить в одном блоке (1024 /ЗАПИСЬ /), и определяет соответствующую константу ЗАП/БЛК (см. также упражнение в конце главы).

Вы можете изменить расположение вновь созданного файла (например, создать несколько различных файлов) путем изменения в строке 5 значения константы ФАЙЛ. Можно изменить и максимальное число блоков, отводимых под ваш файл, заменив число 2 на другое в той же строке. Это значение будет переведено в максимальное число записей путем умножения его на значение, содержащееся в ЗАП/БЛК, и храниться в виде константы МАКС-ЗАП.

де «найти» должен не только осуществляться поиск заданной строки в содержимом полей данного типа, но и «запоминаться» и сама строка, и тип поля, чтобы команды «все» и «еще» смогли бы воспользоваться этой информацией. Можно указать тип поля только одним значением — адресом таблицы полей данного типа. Это означает, что мы можем «запомнить» тип поля, послав его адрес в переменную с помощью слова ЗАПОМНИТЬ. Переменная ТИП служит для обозначения типа поля.

Для того чтобы запомнить *строку*, мы определили буфер с именем ЧТО, куда строка может быть помещена. (Память для буфера определяется в рабочей области PAD, где она может повторно использоваться, и при этом не расходуется память, выделенная под словарь.)

Слово ЗАПОМНИТЬ имеет два назначения: запоминать тип заданного поля в ТИП и заданную строку символов в ЧТО. Если вы посмотрите на слово «найти», обращенное к конечному пользователю, то заметите, что оно в первую очередь посредством слова ЗАПОМНИТЬ запоминает информацию, по которой должен осуществляться поиск, после чего исполняет внутреннее слово —НАЙТИ, осуществляющее по информации, хранимой в ТИП и ЧТО, поиск аналогичной строки. Слова «еще» и «все» тоже используют слово —НАЙТИ, но без слова ЗАПОМНИТЬ. Они осуществляют поиск полей по содержимому, которое было запомнено командой ЗАПОМНИТЬ при последнем применении команды «найти».

Так как с помощью слова «дать» можно получить любую информацию из записи, которую мы уже нашли, применив команду «найти», нам нужен указатель текущей записи. Таким указателем служит переменная ЗАПИСЬ#. Операции, выполняемые в блоке 242 словами ВВЕРХ и ВНИЗ, должны показаться вам тривиальными.

Слово ЗАПИСЬ использует переменную ЗАПИСЬ # для вычисления абсолютного адреса (машинного адреса в буфере на каком-то диске) начала текущей записи. Поскольку ЗАПИСЬ применяет слово **BLOCK**, оно гарантирует, что данная запись действительно *существует* в буфере.

Обратите внимание на то, что ЗАПИСЬ допускает расположение одного файла в нескольких смежных блоках. **/MOD** делит значение, находящееся в переменной ЗАПИСЬ#, на число записей в одном блоке (в нашем случае 16, так как каждая запись имеет длину в 64 байта). Частное указывает тот блок относительно первого блока, где должна находиться обрабатываемая запись, а остаток определяет смещение этой записи относительно начала вычисленного блока.

В таблице полей содержатся *относительные* значения адреса поля и длины. Однако нам для таких слов, как **TYPE**, **MOVE** и

—**TEXT**, часто требуется знать абсолютные адрес и длину. Поэтому покажем, как в определении слова ПОЛЕ адрес таблицы полей преобразуется в абсолютные адрес и длину, а затем как в определении слова .ПОЛЕ используется слово ПОЛЕ.

Слово ПОМЕСТИТЬ с помощью слова ПОЛЕ вычисляет адрес и счетчик поля назначения для слова ЧТЕНИЕ. Последнее выбирает из входного потока строку, ограниченную запятой, и помещает эту строку в поле назначения. Заметьте, что ЧТЕНИЕ инициирует слово ПОДРОВНЯТЬ, которое введено для урезания числа символов перекачиваемой строки в том случае, если источник превышает поле назначения.

В определении слова ПУСТАЯ в блоке 243 обращают на себя внимание два обстоятельства. Первое из них — это способ обнаружения пустой записи. Если первый байт какой-либо записи не содержит информации, то можно считать, что вся запись пуста (в этом заключается принцип выполнения слова «внести»). Если в первом байте содержится некоторый символ, значение которого в коде ASCII меньше 33 (код 32 соответствует пробелу), значит, в данном байте находится невидимый символ и, следовательно, строка пуста. В пустом блоке могут находиться нули или он может быть полностью заполнен пробелами, но в любом случае такие записи будут рассматриваться как пустые. По обнаружении пустой записи **LEAVE** завершает цикл. В переменной ЗАПИСЬ # находится номер свободной записи.

Второе обстоятельство, связанное со словом ПУСТАЯ, заключается в том, что это слово прерывает выполнение посредством **ABORT** при заполнении файла, т. е. если оно прошлось по всем записям и не обнаружило среди них ни одной пустой. Для того чтобы обойти все записи, можно воспользоваться циклом **DO**, но как узнать по окончании выполнения цикла, была ли обнаружена во время просмотра хотя бы одна пустая запись?

Прежде чем приступить к выполнению цикла, целесообразно оставить в вершине стека значение истины, которое будет служить флагом. При обнаружении пустой записи мы можем сменить значение флага на нуль (посредством слова **NOT**) перед тем, как выйти из цикла. Если после выхода из цикла в вершине стека по-прежнему находится единица, значит, пустая запись не обнаружена, а если нуль, то обнаружена.

Мы применяем аналогичный прием в определении слова —НАЙТИ. Это слово оставляет в вершине стека флаг для слова, которое его выполняет, а именно: «найти», «еще», «все» или (ПАРА). Флаг показывает, была ли найдена заданная строка до конца файла. Каждое из перечисленных слов внешнего уровня в зависимости от состояния флага должно принимать соответствующее решение. Если заданная строка не найдена, то в вершине стека будет значение истины (отсюда и название слова —НАЙТИ).

С учетом контекста использования слова —НАЙТИ изменим значения флага на обратные. Так как значение флага должно быть истинным в том случае, когда поиск заданной строки окончился неудачей, проще всего определить это слово таким образом, чтобы до начала поиска в вершине стека была единица, которая заменялась бы нулем лишь при успешном завершении поиска. Обращаем ваше внимание на то, что во время выполнения цикла в вершине стека находятся два значения: только что рассмотренный флаг и адрес таблицы полей, определяющей поле, по которому ведется поиск. Поскольку адрес нам требуется на каждом шаге выполнения цикла, а значение флага, возможно, понадобится всего один раз, мы решили хранить адрес в вершине стека, а флаг — под ним. Для этого мы и использовали выражение

```
SWAP NOT SWAP
```

Между прочим мы могли бы избежать возникновения такой ситуации, если бы вместо выражений

```
ТИП @

DUP ПОЛЕ
```

перед циклом и внутри его для того, чтобы обеспечить оба этих значения в вершине стека, мы применили бы выражение

```
ТИП @ ПОЛЕ
```

внутри цикла. Мы отказались от этого потому, что всегда стремимся минимизировать число операций, выполняемых внутри цикла: операции внутри цикла повторяются многократно и занимают очень много времени.

На этом мы завершаем описание программы в целом и надеемся, что вы без труда разберетесь в деталях ее работы самостоятельно по приводимому ниже листингу[1].

---

[1] *Для пользователей систем фиг-Форта.* Прежде чем загрузить программу с файловой системой, убедитесь в том, что приведенные ниже определения скомпилированы *первыми*.

```
: VARIABLE   0 VARIABLE ;
: CREATE   <BUILDS DOES> ;
: BLANK  ( a # -- ) BLANKS ;
: WORD  ( -- a)   WORD  HERE ;
: >IN  ( -- a)   IN ;
```

```
561 LIST

HELP  описание директив ФАЙЛОВОЙ СИСТЕМЫ.
внести пополняет базу данных.Данные вводятся в четыре поля текущей записи
  следите за правильным использованием запятых и пробелов.
  Использование:  внести Рэйзер,Дан,диктор,555-1212
удалить  заполнение пробелами текущей записи и обновление дискового буфера.
изменить  размещение входного текста в заданном поле текущей записи.
  Использование: изменить работа программист
найти  поиск входного фрагмента в памяти и индикация его наличия или отсут-
  ствия.  Использование: найти имя Дан
дать  выдача данных из указанного поля текущей записи.
  Использование: дать телефон
еще  выдача следующего найденного фрагмента за ЗАПИСЬ# .
все  выдача всех искомых полей базы данных
пара  поиск записи по содержимому двух полей.
Использование: пара работа диктор,телефон 555-9876
фио  поиск по имени и фамилии. Использование: фио Рэйзер,Дан
```

```
562 LIST
Текст в коде ASCII  запоминается на диск в очередную запись длиной в одну
 строку, на которую указывает ЗАПИСЬ#, с возможностью доступа к
 четырем полям этой записи по именам полей. Для поиска в базе
 данных по содержимому входного буфера применяется слово -TEXT.
 Удачный поиск завершается выходом из цикла посредством EXIT и установкой
 ЗАПИСЬ# на вывод содержимого поля, определяемого переменной ТИП.
Имя каждого поля содержит смещение относительно начала текущей
 записи и счетчик (длину) в байтах. С помощью этих имен мы можем
 в слове ПОЛЕ для доступа к данным выдавать виртуальные адреса.
 Мы можем осуществлять доступ к полям заданной текущей записи (ЗАПИСЬ#)
 по имени поля, а затем работать в окрестности полученного адреса.
Значение ЗАПИСЬ# последовательно увеличивается.
ТИП содержит адрес интересующего нас поля; используется для
 входа в запись, на которую указывает ЗАПИСЬ# .
ЧТО является буфером, содержащим входной тест, по которому
 осуществляется поиск в базе данных.
```

```
563 LIST

ВВЕРХ  устанавливает указатель записи в начало базы данных.
ВНИЗ  устанавливает указатель на следующую запись.
ПОДРОВНЯТЬ  устанавливает счетчик для CMOVE в пределах длины
 поля.
ЧТЕНИЕ  заполняет буфер пробелами и заносит в него указанный
 счетчик байтов. Ограничителем поля является знак "," в коде ASCII.
ЗАПИСЬ  оператор, работающий с виртуальной памятью и
 вырабатывающий адрес внутри дискового блочного буфера. Этот
  адрес является началом текущей записи.
ПОЛЕ  выбирает смещение и длину в одной из четырех переменных
  для получения адреса и счетчика, необходимых при выводе информации.
ПОМЕСТИТЬ  помещает символы, введенные с клавиатуры, в
 обновляемый дисковый блочный буфер.
```

Затем замените выражение блока 243

```
ABORT" Переполнение файла"
```

на следующее:

```
IF ." Переполнение файла"  QUIT  THEN
```

240 LIST

```
( Простая файловая система )   DECIMAL
: HELP   SCR @  561 LIST  SCR ! ;  HELP
241 243 THRU
: внести   ПУСТАЯ  фамилия ПОМЕСТИТЬ  имя ПОМЕСТИТЬ  работа ПОМЕСТИТЬ
   телефон ПОМЕСТИТЬ ;
: удалить   ЗАПИСЬ /ЗАПИСЬ BLANK  UPDATE ;
: изменить   '   >BODY ПОМЕСТИТЬ ;
: найти   ( поле текст)   '   >BODY ЗАПОМНИТЬ  ВВЕРХ -НАЙТИ  IF
      ОТСУТСТВУЕТ  ELSE  .ИМЯ  THEN ;

: дать ( поле)   '   >BODY .ПОЛЕ ;
: еще   ВНИЗ  -НАЙТИ  IF ." Больше нет " ELSE  .ИМЯ  THEN
: все   ВВЕРХ BEGIN  CR  -НАЙТИ NOT  WHILE  .ИМЯ  ВНИЗ  REPEAT ;

: пара    '   >BODY ЗАПОМНИТЬ   '   >BODY PAD 80 ЧТЕНИЕ  имя (ПАРА) ;
: фио   фамилия ЗАПОМНИТЬ  PAD 80 ЧТЕНИЕ  имя (ПАРА) ;
```

241 LIST

```
     ( Поля)
VARIABLE ЗАПИСЬ#    ( текущая запись)
VARIABLE ТИП ( указатель в таблицу полей на последнее используемое поле)
: ЧТО ( - a)   PAD 100 + ;
                   ( смещение)      ( длина)
CREATE фамилия        0 ,            16 ,
CREATE имя           16 ,            12 ,
CREATE работа        28 ,            24 ,
CREATE телефон       52 ,            12 ,

64 CONSTANT /ЗАПИСЬ                  ( число байт в одной записи)
1024 CONSTANT /БЛОК                  ( число байт в одном блоке)
/БЛОК /ЗАПИСЬ / CONSTANT ЗАП/БЛК      ( число записей в блоке)
244 CONSTANT ФАЙЛЫ                   ( с данного блока начинаются файлы)
2  ( блоки)  ЗАП/БЛК * CONSTANT МАКС-ЗАП ( максимальное число записей)
```

242 LIST

```
   ( Записи)
: ВВЕРХ    0 ЗАПИСЬ# ! ;
: ВНИЗ    1 ЗАПИСЬ# +! ;
: ПОДРОВНЯТЬ ( a # a #)    >R  SWAP  R> MIN  CMOVE ;
: ЧТЕНИЕ ( a #)   2DUP BLANK  ASCII , WORD COUNT
  2SWAP ПОДРОВНЯТЬ ;
: ЗАПИСЬ ( - a)   ЗАПИСЬ# @ ЗАП/БЛК /MOD ФАЙЛЫ + BLOCK

  SWAP /ЗАПИСЬ * + ;

: ПОЛЕ ( a - a' n)   2@ ЗАПИСЬ +  SWAP ;

: ПОМЕСТИТЬ ( a)   ПОЛЕ ЧТЕНИЕ UPDATE ;
```

а в блоке 240 замените каждое вхождение выражения ' >BODY на [COMPILE]'. Наконец, загрузите необходимые дополнительные команды, указанные в примечании 3.

2. *Для пользователей систем полиФорта.* Перед загрузкой программы работы с **файлами убе**дитесь, что вы сначала скомпилировали следующее определение:

```
: >BODY   ;
```

```
564 LIST
.ПОЛЕ  вывод информации из заданного поля текущей записи.
.ИМЯ  вывод имени и фамилии из заданной записи.

ЗАПОМНИТЬ  устанавливает,ЧТО      собой представляет текстовый
 фрагмент и ТИП поля, по которому будет осуществляться поиск.
ПУСТАЯ  установка указателя записи на следующую доступную
 (свободную) запись; если достигнуто значение МАКС-ЗАП,то ABORT.
 Запись,первый байт которой равен 32 (пробел) или 0, считается пустой.
-НАЙТИ  побайтное сравнение фрагмента из ЧТО с содержимым выбранного
  поля каждой записи. Если сравнение успешное или цикл поиска завер-
  шен, выдается флаг "истина". Это логическое значение используется
  словами (ПАFA), найти, все и еще.
(ПАРА)  по адресам двух полей выбираются два текстовых фрагмента и
  сравниваются с содержимым выбранных полей каждой записи.
```

```
565 LIST
```

```
566 LIST
```

и загрузили все необходимые определения, указанные в примечании 3.

3. *Для пользователей систем, в которых нет следующих слов.* Загрузите, если нужно, приведенные ниже определения (после того, как осуществится загрузка определений, указанных в примечаниях 1, 2):

```
-1 CONSTANT TRUE
: ASCII  ( -- c)  BL WORD 1+ C@ [COMPILE] LITERAL ; IMMEDIATE
: \   IN @  64 /  1+  64 *  IN ! ;
: -TEXT  ( a1 # a2 -- ?)  2DUP + SWAP DO  DROP 2+
     DUP 2- @  I @ - DUP  IF DUP ABS / LEAVE THEN
     2 +LOOP  SWAP DROP ;
: TEXT  ( c)  PAD 80 BLANK  WORD COUNT  PAD SWAP CMOVE ;
```

```
243 LIST
  ( Выдача информации)
: .ПОЛЕ  ( a)   ПОЛЕ -TRAILING TYPE  SPACE ;
: .ИМЯ   имя .ПОЛЕ  фамилия .ПОЛЕ ;

: ЗАПОМНИТЬ   ( a)   DUP ТИП !  2+ @  ASCII , TEXT
   PAD ЧТО ROT CMOVE ;

: ПУСТАЯ   TRUE  МАКС-ЗАП 0 DO  I ЗАПИСЬ# ! ЗАПИСЬ C@ 33 < IF
   NOT LEAVE  THEN  LOOP  ABORT" Переполнение файла" ;
: -НАЙТИ ( - t)   TRUE  ТИП @  МАКС-ЗАП ЗАПИСЬ# @ DO
   I ЗАПИСЬ# !  DUP ПОЛЕ ЧТО -TEXT 0= IF
     SWAP NOT SWAP  LEAVE  THEN  LOOP  DROP ;
: ОТСУТСТВУЕТ   ." Сведения отсутствуют " ;
: (ПАРА)  ( a)   МАКС-ЗАП 0 DO  I ЗАПИСЬ# !  -НАЙТИ IF
   ОТСУТСТВУЕТ LEAVE  ELSE  DUP ПОЛЕ PAD -TEXT 0= IF
     .ИМЯ LEAVE  THEN  THEN  LOOP  DROP ;

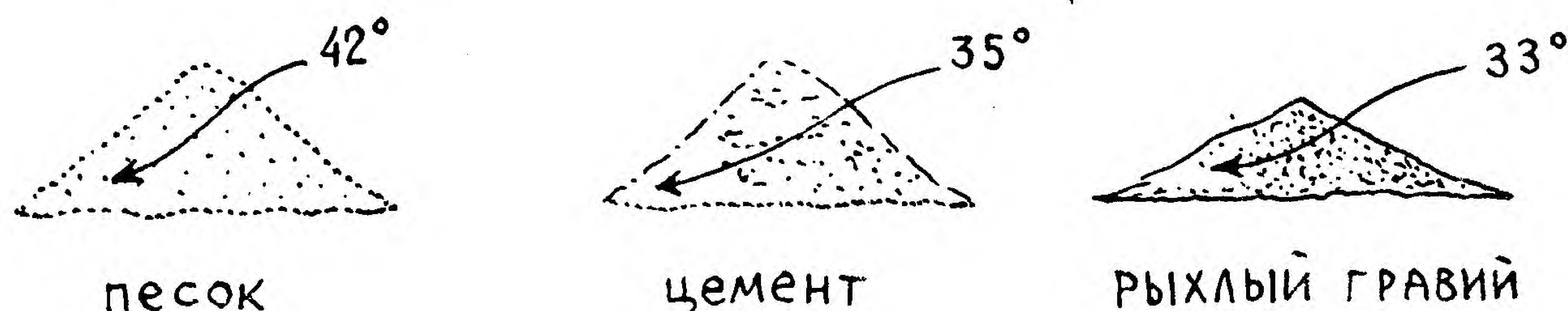
244 LIST
Филлмор        Миллард      президент                  нет телефона
Линкольн       Авраам       президент                  нет телефона
Бронте         Эмилия       писатель                   нет телефона
Рэйзер         Дан          диктор                     555-9876
Фитцджеральд   Элла         певец                      555-6789
Чанг           Конни        диктор                     555-9653
Маккартни      Пол          композитор                 555-1212
Вашингтон      Джордж       президент                  нет телефона
Рейнольдз      Фрэнк        диктор                     555-7865
Силлз          Беверли      оперная звезда             нет телефона
Форд           Генри        капиталист                 нет телефона
Дьюхерст       Коллин       актриса                    555-1234
Уандер         Стив         композитор                 555-0097
Фуллер         Бакминстер   известный архитектор       555-3456
Ролз           Джон         философ                    555-4567
Трюдо          Гарри        юморист                    555-4321

245 LIST
Ван Еарен      Абигейл      обозреватель               555-2233
Симончик       Елена        программист                нет телефона
Томпсон        Хантер.С     журналист                  555-9452
Маркузе        Герберт      философ                    555-1849
Джаббар        Карим Абдул  баскетболист               555-4721
Маги           Трэйвис      вымышленный детектив       555-8792
Дидьон         Джоан        писатель                   555-0089
Фразетта       Фрэнк        артист                     555-9091
Хенсон         Джим         кукловод                   555-0101
Саган          Карл         астроном                   555-7070
```

БЕЗ ВЗВЕШИВАНИЯ

В качестве второго примера приведем математическую задачу, решение которой, по мнению многих, невозможно без привлечения операций над числами с плавающей точкой. Мы покажем вам, как арифметика с фиксированной точкой позволяет решать довольно сложные уравнения, и при этом ни диапазон, ни точность представления чисел не ухудшаются.

Вычислим вес конусообразной «кучи» некоторого материала, зная ее высоту и угол откоса, а также плотность материала. Чтобы сделать нашу задачу более конкретной, давайте взвесим кучи песка, гравия и цемента. Крутизна каждой кучи, называется *углом естественного откоса,* зависит от вида материала. Например, песчаные кучи более крутые, чем из гравия.



(На самом деле эти величины колеблются в большом диапазоне в зависимости от разных факторов. Для иллюстрации мы выбрали приблизительные значения угла откоса и плотности.)

Существует формула вычисления массы конусообразной кучи высотой h (в футах), углом естественного откоса θ (в градусах) и плотностью материала D (в фунтах на кубический фут)[1]:

$$W = \frac{\pi h^3 D}{3\,\mathrm{tg}^2(\theta)}.$$

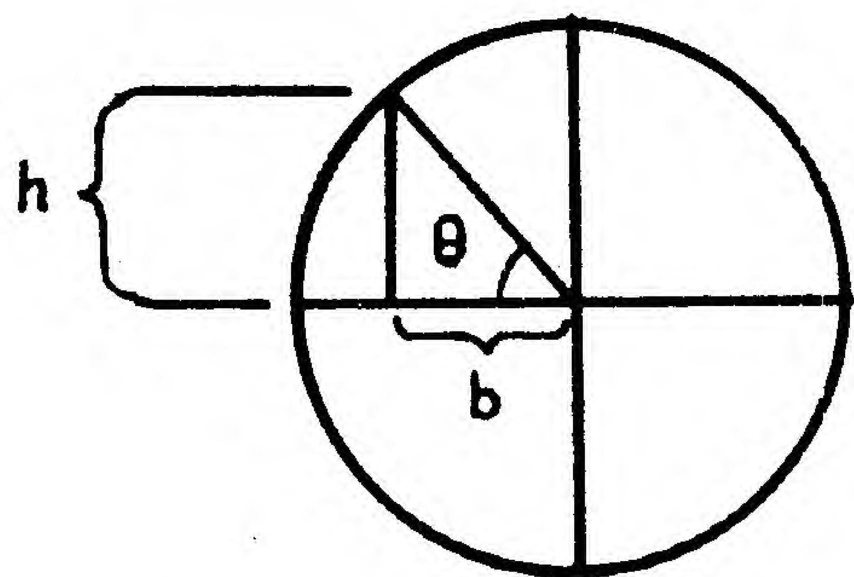
Запишем эту формулу на Форте. Условимся считать, что аргументы нашей программы будут вводиться в такой последовательности: название материала, например ПЕСОК, а затем высота кучи. В результате выполнения программы мы должны получить массу кучи сухого песка. Допустим, что для любого материала его плот-

---

[1] *Для скептиков.* Объем конуса V вычисляется по формуле

$$V = \frac{1}{3}\pi b^2 h,$$

где b — радиус основания; h — высота. Мы можем найти радиус основания, зная угол откоса, или, более точно, тангенс этого угла. Тангенсом некоторого угла называется отношение катета противолежащего (на рисунке h) к катету прилежащему (на рисунке b):

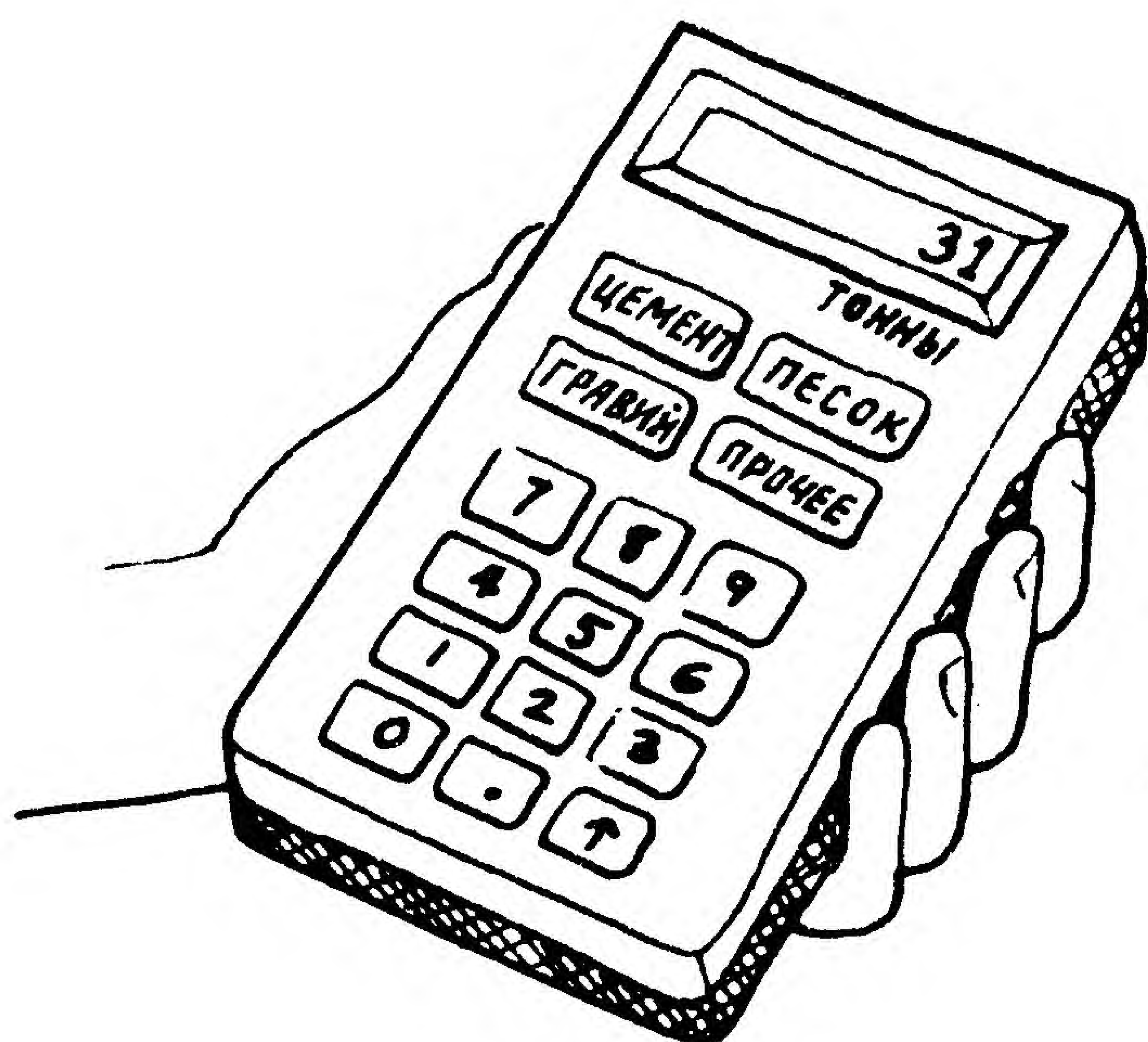


Если обозначить этот угол через θ, то $tg\,\theta = h/b$. Следовательно, $b = h/tg\,\theta$. Подставив это выражение в формулу для V и затем умножив результат на плотность D (в фунтах на кубический фут), получим приведенную выше формулу.

ность и угол естественного откоса остаются неизменными. Поэтому можно записать указанные две величины для каждого вида материала в некоторую таблицу. Так как в конечном итоге нам потребуется тангенс угла, а не его величина в градусах, будем хранить значение тангенса. Например, угол естественного откоса для цемента составляет 35°, а его тангенс равен .700. Мы будем хранить это значение в виде целого 700.

ЦЕМЕНТ
131
700

Помните, что наша цель заключается не в том, чтобы просто получить результат. Мы составляем такую программу, по которой компьютер или иное вычислительное устройство выдает ответ самым быстрым, самым эффективным и самым точным способом. Как уже отмечалось в гл. 5, для записи математических выражений посредством арифметики с фиксированной точкой вам придется приложить немало усилий. Но вы будете вознаграждены за свои страдания. Во-первых, значительно увеличится скорость выполнения, что очень важно в тех случаях, когда какой-нибудь процесс разбивается на миллионы шагов или когда ежеминутно приходится производить несколько тысяч операций. Во-вторых, уменьшится размер программы и вы сможете, например, выполнить ее с помощью калькулятора, специально предназначенного для вычисления веса материала. Форт часто применяется в такого рода машинках.



Начнем решение задачи с установления порядка величин. Высота всех трех куч лежит в пределах от 5 до 50 футов. Вычислив массу кучи цемента, мы получим 35 000 000 фунтов. Но так как кучи не имеют форму правильного конуса, а мы берем средние значения величин, точность вычислений не может превышать четырех

или пяти порядков[1]. Если перевести наш результат в тонны, то получим 17 500 т. Это значение вполне удовлетворяет представлению числа одинарной длины, так что в нашей программе условимся применять арифметические операции только над значениями одинарной длины.

Те программы, в которых требуется бо́льшая точность, могут быть написаны с использованием представления чисел двойной длины. Как вы увидите позднее, для сравнения мы даже создадим второй вариант нашей программы, где будут задействованы арифметические операции над 32-разрядными числами. Пока же мы хотим показать, что требуемую точность можно обеспечить, применяя операции Форта только с 16-разрядными значениями.

Выполнив вычисления для кучи высотой 40 футов, мы обнаружили, что изменение высоты на одну десятую может привести к изменению ее массы на 25 тонн. Поэтому входные данные будем исчислять не в целых футах, а в десятых долях фута. Желательно, чтобы пользователь имел возможность вводить следующее выражение:

```
15 ФУТОВ  2 ДЮЙМОВ  КУЧА
```

где слова ФУТОВ и ДЮЙМОВ представляли бы футы и дюймы с точностью до десятых долей дюйма, а слово КУЧА производило бы необходимые вычисления. Слова ФУТОВ и ДЮЙМОВ можно было бы определить так:

```
: ФУТОВ   ( футы -- вес-в-масштабе)  10 * ;
: ДЮЙМОВ   ( вес-в-масштабе -- вес-в-масштабе' )
    100 12 */  5 +  10 /  + ;
```

Использование слова ДЮЙМОВ не обязательно. Таким образом, выражение 23 ФУТОВ поместит в вершину стека число 230, выражение 15 ФУТОВ 4 ДЮЙМОВ — число 153 и т. д. (Между прочим довольно легко организовать ввод данных и в десятых долях дюйма с десятичной точкой, например: 15.2. В таком случае слово **NUMBER** переводит вводимое число в значение двойной длины. Так как мы имеем дело только с числами одинарной длины, для того чтобы удалить старший байт, достаточно просто применить слово **DROP**.)

При написании определения КУЧА нужно попытаться обеспечить максимальную точность, не выходя за границы 15 разрядов. По нашей формуле первое, что требуется сделать, — возвести аргумент в куб. Однако напомним, что аргументом может служить

[1] *Для специалистов-математиков.* На самом деле, так как высота в нашем примере выражается тремя цифрами, мы не можем ожидать точности большей, чем три порядка. Однако в учебных целях мы выберем точность, превышающую четыре порядка.

значение высоты вплоть до 50 футов, т. е. при выбранном масштабе 500. Даже если мы возведем это значение лишь *в квадрат*, то получим 250 000, что уже превышает возможности представления одинарной точности. Но для того чтобы выразить ответ в тоннах, рано или поздно в ходе вычислений нам придется выполнить деление на 2000, поэтому выражение

```
DUP DUP 2000 */
```

будет одновременно возводить аргумент в квадрат и переводить в тонны, так как под промежуточный результат в операции */ отводится слово двойной длины. Если наш аргумент равен 500, то в результате получится 125.

Высота кучи может оказаться равной всего лишь пяти футам. Возведение этого значения в квадрат дает 25, а при делении последнего числа на 2000 средствами целочисленной арифметики мы получим нуль, что свидетельствует о неудачном выборе масштаба с небольшими значениями аргумента. Для обеспечения максимальной точности мы не должны при масштабировании получать меньшие значения, чем требуется. Число 250 000 вполне представимо средствами арифметики одинарных чисел, если его предварительно разделить на 10, поэтому мы начнем наше определение КУЧА следующим выражением:

```
DUP  DUP 10 */
```

На данном этапе результат с учетом масштабирования окажется в десять раз большим (25000 вместо 2500.0), чем требуется.

Далее необходимо возвести аргумент в *куб*. Непосредственное умножение опять приведет к результату двойной длины и, значит, масштабирование должно производиться с применением операции */. Как видите, выбрав в качестве делителя 1000, мы продолжаем оставаться в пределах одинарной длины. Теперь наш результат будет в десять раз меньшим (12 500 вместо 125 000), чем требуется, и все еще сохраняется точность в пять знаков. Согласно приведенной выше формуле нужно умножить аргумент на *pi*. Вы знаете, что на Форте это можно сделать с помощью следующего выражения:

```
355 113 */
```

Кроме того, мы должны разделить наш аргумент на три. Оба указанных действия можно осуществить посредством выражения

```
355 339 */
```

что не вызывает никаких проблем с масштабированием. Затем полученное выражение делим на квадрат тангенса (путем последовательного *двойного* деления на значение тангенса). Поскольку значение тангенса, хранимое в таблице, увеличено в 1000 раз, чтобы выполнить деление, нужно делимое умножить на 1000 и разделить на табличное значение тангенса:

```
1000 ТЕТА @ */
```

Такое действие мы должны произвести дважды, поэтому оформим его в виде определения с именем /TAN (для деления-на-тангенс) и применим это определение дважды в определении слова КУЧА. Наш результат пока еще будет в десять раз меньше, чем на самом деле (26711 вместо 267110 при максимальных значениях аргументов).

Нам остается лишь умножить полученное значение на величину плотности материала, максимальное значение которой равно 131 фунт на кубический фут. Во избежание переполнения уменьшаем плотность в 100 раз, применяя выражение

```
ПЛОТНОСТЬ @ 100 */
```

Проверив выполнение программы в этой точке с данными о куче цемента высотой 50 футов, получим число 34991, что превышает 15 разрядов. Теперь пора принять во внимание значение 2000. Вместо

```
ПЛОТНОСТЬ @ 100 */
```

мы можем написать:

```
ПЛОТНОСТЬ @ 200 */
```

и наш ответ будет приведен к целому числу тонн.

Этот вариант программы вы найдете на распечатке блока 246, которая приводится ниже. Как уже упоминалось ранее, в блоке 248 находится вариант той же программы с использованием арифметических операций над данными двойной длины. Здесь вы вводите высоту в виде числа двойной точности с одним знаком после точки, представляющим десятые доли фута, например 50.0 футов, а далее следует слово ФУТЫ.

Выполняя целочисленные арифметические операции над числами двойной длины, мы в состоянии округлять полученный вес кучи до *ближайшего* целого фунта. В нашем случае применение целочисленной арифметики двойной длины открывает не меньшие

возможности, чем те, которые предоставляет более мощная арифметика с плавающей точкой. Ниже для сравнения приводятся результаты, полученные с помощью калькулятора, имеющего диапазон представления чисел в 10 десятичных цифр, средств Форта одинарной длины и средств Форта двойной длины. Результаты представлены для кучи цемента высотой 50 футов с использованием табличных значений:

	В ФУНТАХ	В ТОННАХ
калькулятор	34.995.634	17.497.817
16-разрядный Форт	–	17.495
32-РАЗРЯДНЫЙ Форт	34.995.634	17.497.817

Результаты же работы нашей программы таковы:

```
246 LOAD ok              ( компиляция варианта с одинарной длиной)
ЦЕМЕНТ ok
10 ФУТОВ КУЧА = 138 тонн цемента ok
10 ФУТОВ 3 ДЮЙМОВ КУЧА = 151 тонн цемента ok
СУХОЙ-ПЕСОК
10 ФУТОВ КУЧА = 81 тонн песка ok

248 LOAD ok              ( компиляция варианта с двойной длиной)
ЦЕМЕНТ ok
10.0 ФУТОВ = 279939 фунтов цемента или 139.969 тонн ok
```

Замечание по поводу компиляции строк. Определение слова МАТЕРИАЛ требует трех аргументов для каждого материала, одним из которых является адрес некоторой строки. Слово .МАТЕРИАЛ с помощью этого адреса выводит название рассматриваемого материала. Для помещения этой строки в словарь и передачи адреса слову МАТЕРИАЛ мы определили слово с именем ," (запятая-кавычки).

В первую очередь оно вносит в вершину стека значение **HERE** для слова МАТЕРИАЛ, так как именно по данному адресу будет компилироваться строка, а затем инициирует слово **STRING** для компиляции строкового литерала, ограниченного двойной кавычкой. В некоторых системах это слово можно определить следующим образом:

```
: STRING  ( c)  WORD C@ 1+ ALLOT ;
```

```
Block# 246
\ Определение массы конической кучи -- одинарная длина
VARIABLE ПЛОТНОСТЬ  VARIABLE ТЕТА  VARIABLE Н.М.
: ,"  ( --a)  HERE  ASCII " STRING ;
: .МАТЕРИАЛ   Н.М. @  COUNT TYPE SPACE ;
: МАТЕРИАЛ   ( 'строка плотность тета -- )  CREATE , , ,
     DOES>  DUP @ ТЕТА !  2+ DUP @ ПЛОТНОСТЬ !  2+ @ Н.М. ! ;

: ФУТОВ  ( футы -- вес-в-масштабе)    10 * ;
: ДЮЙМОВ  ( вес-в-масштабе -- вес-в-масшбабе')
    100 12 */  5 +  10 /  + ;

: /TAN  ( n -- n')  1000 ТЕТА @  */ ;
: КУЧА  ( вес-в-масштабе -- )
    DUP DUP 10 */  1000 */   355 339 */  /TAN /TAN
    ПЛОТНОСТЬ @  200 */  ." = "  .  ." тонн "  .МАТЕРИАЛ ;
247 LOAD

Block# 247
\ Таблица материалов
\ адрес-строки                 плотность   тета
," цемента"                      131        700   МАТЕРИАЛ ЦЕМЕНТ
," рыхлого гравия"                93        649   МАТЕРИАЛ РЫХЛЫЙ-ГРАВИЙ
," плотного гравия"              100        700   МАТЕРИАЛ ПЛОТНЫЙ-ГРАВИЙ
," сухого песка"                  90        754   МАТЕРИАЛ СУХОЙ-ПЕСОК
," мокрого песка"                118        900   МАТЕРИАЛ МОКРЫЙ-ПЕСОК
," глины"                        120        727   МАТЕРИАЛ ГЛИНА




 ЦЕМЕНТ



Block# 248
\ Масса конической кучи -- двойная длина
VARIABLE ПЛОТНОСТЬ   VARIABLE ТЕТА   VARIABLE Н.М.
: ,"   ( — a)  HERE  ASCII " STRING ;
: .МАТЕРАЛ   Н.М. @  COUNT TYPE SPACE ;
DU.3  ( du -- )  <#  # # #  ASCII . HOLD  #S  #> TYPE SPACE ;
МАТЕРИАЛ   ( 'строка плотность тета -- )  CREATE , , ,
  DOES>  DUP @ ТЕТА !  2+ DUP @ ПЛОТНОСТЬ !  2+ @ Н.М. ! ;
: КУБ  ( d -- d')  2DUP  OVER  10 M*/  DROP  10 M*/ ;
: /TAN  ( d -- d')  1000 ТЕТА @  M*/ ;
: ФУТОВ  ( d -- )  КУБ  355 339 M*/  ПЛОТНОСТЬ @  1 M*/
   /TAN /TAN  5 M+1  10 M*/
   2DUP  ." = "  D.  ." фунтов  "  .МАТЕРИАЛ
   1 2 M*/  ." или " DU.3  ." тонн " ;
247 LOAD
```

ФОРТ-АССЕМБЛЕР

Форт часто используют в прикладных областях, где от программ требуется высокая скорость выполнения, например, при обработке сигналов информация поступает в реальное время и компьютер должен справляться с ее обработкой. Как правило, вы существенно выигрываете в скорости, если работаете с Фортом, а не с другими языками программирования, но языку Ассемблера он все же в этом отношении уступает. (Вновь создаваемые Форт-процессоры, такие, как NOVIX NC 4000, непосредственно вы-

полняют команды Форта высокого уровня быстрее, чем традиционные процессоры свои машинные команды. Ассемблер, описанный в данном разделе, имеет смысл только для систем, функционирующих на обычных процессорах.)

В прикладной программе почти все время выполнения приходится лишь на ее небольшую часть, а именно на так называемые *внутренние циклы*. Если ваша программа, написанная на Форте — языке высокого уровня, работает слишком медленно, вы можете значительно ускорить ее выполнение, переписав один или два внутренних цикла на языке Ассемблера.

В большинстве языков высокого уровня нет хороших средств автономного создания программ на ассемблере и соединения их с основной программой. На Форте же это обычный процесс. Определения на ассемблере Форта выглядят почти так же, как и определения высокого уровня. Если тело определения через двоеточие содержит код высокого уровня:

```
: НОВОЕ-ИМЯ   ( код высокого уровня . . .) ;
```

то тело ассемблерного определения включает команды на языке Ассемблера:

```
CODE НОВОЕ-ИМЯ  ( код на языке ассемблера . . .)  END-CODE
```

(Слово завершения ассемблерного кода варьируется от системы к системе; Стандарт-83 рекомендует **END-CODE**.)

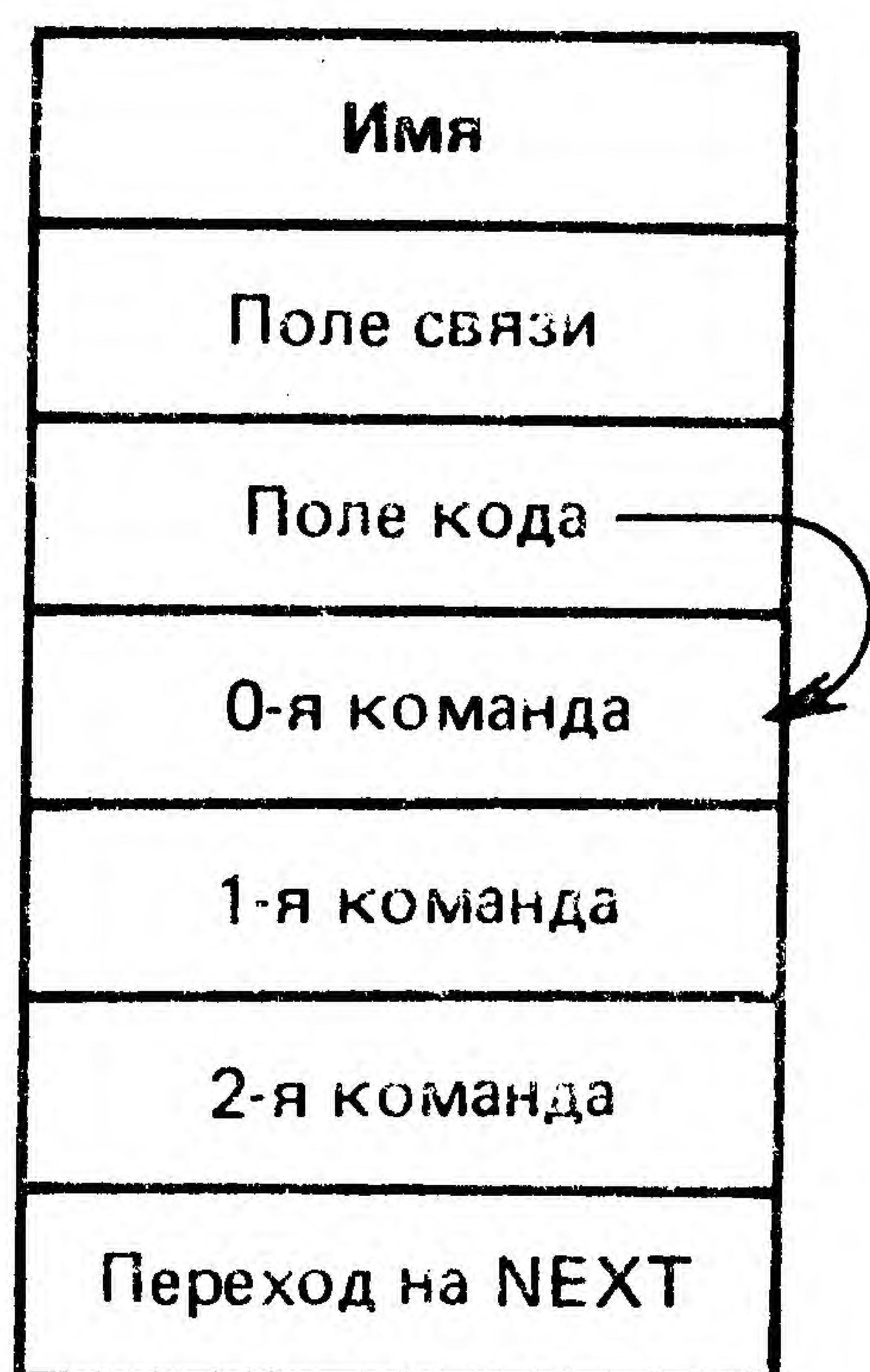
Слово, которое определяется посредством **CODE**, хранится в словаре так же, как и все остальные, и выполняется или вызывается подобно любому другому слову. Определенное соответствующим образом это слово будет выполняться со значениями из стека, поэтому вы можете передавать ему аргументы так, как если бы оно было определено через двоеточие. На самом деле, выполняя некоторое слово, вы не в состоянии установить, определено ли оно через двоеточие или через **CODE** (разве что по скорости выполнения).

Лучше всего писать программу на языке высокого уровня. После того как она начала правильно работать, вы можете выявить те участки, на выполнение которых тратится основное время и переписать их в машинных кодах. Повторно откомпилируйте программу, и она будет работать намного быстрее. Альтернативный способ — заранее фиксировать критичные по времени участки — не столь эффективен.

Рассмотрим создание конкретного ассемблера на примере ассемблера 8080. Очевидно, что для каждого процессора должен существовать свой ассемблер и что ассемблер 8080 подходит только для данного процессора. Если вы введете в ваш компьютер при-

веденный здесь пример на ассемблере, то получите сгенерированный машинный код для 8080, что, собственно, вам и требуется. Но попытавшись полученный код *выполнить*, вы потерпите неудачу. Наш пример показывает, как легко писать на Форт-ассемблере, объясняет основные принципы разработки ассемблера и демонстрирует мощь определяющих слов Форта.

Начнем с определяющего слова **CODE**. Его назначение — создание заголовка словарной статьи, которая при выполнении передаст управление по адресу, содержащему машинный код. Выполнить это проще, чем понять. Вспомните (см. гл. 9), что все определения снабжены полем кода, которое указывает машинный код. В определении **CODE** такой указатель должен указывать поле параметров данного определения:



Таким образом, простое определение слова **CODE** может иметь вид:

```
: CODE   CREATE  HERE  HERE 2- ! ;
```

Теперь нам нужен набор слов, позволяющий осуществлять трансляцию машинных команд в словарь. Для начала выберем несложное слово. Команда процессора 8080 CMA вычисляет дополнение содержимого регистра A. Код этой операции в двоичной системе счисления выглядит так: 00101111. Чтобы транслировать команду, введем следующее определение:

```
HEX
: CMA   2F C, ;
```

Еще одна простая команда XCHG обеспечивает обмен содержимым между парами регистров D — E и H — L. Код такой операции: 11101011. Далее мы можем ввести определение:

```
: XCHG   EB C, ;
```

Подведем предварительные итоги. Мы задали себе синтаксис написания определений машинных команд и предыдущими действиями создали средства их спецификации. Теперь можно ввести следующий текст:

```
CODE TECT   CMA   XCHG . . .
```

Получено слово ТЕСТ, выполняющее машинные команды СМА и XCHG. Вы можете для проверки этого слова воспользоваться словом **DUMP** (но ни в коем случае не инициируйте слово ТЕСТ!).

Как заканчивается CODE-определение, мы покажем позднее, а пока вернемся к определению машинных команд. У нас уже определены две команды, состоящие из восьмиразрядного кода операции. Процессор 8080 имеет довольно много команд такого типа. Поэтому нам необходимо слово для определения всех подобных команд (назовем их командами типа1) 1MI.

```
: 1MI   ( код-операции -- )   CREATE C,   DOES> C@ C, ;
```

Определим с помощью введенного слова следующие команды (первые два определения по-новому создают уже имеющиеся у нас команды):

```
HEX
2F 1MI CMA     EB 1MI XCHG     00 1MI NOP      76 1MI HLT
F3 1MI DI      FB 1MI EI       07 1MI RLC      0F 1MI RRC
17 1MI RAL     1F 1MI RAR      E9 1MI PCHL     F9 1MI SPHL
E3 1MI XTHL    27 1MI DAA      37 1MI STC      3F 1MI CMC
C0 1MI RNZ     C8 1MI RZ       D0 1MI RNC      D8 1MI RC
E0 1MI RPO     E8 1MI RPE      F0 1MI RP       F8 1MI RM
C9 1MI RET
```

Определяющее слово 1MI создает семейство команд, каждую из которых отличает уникальный код операции, но при компиляции все они ведут себя одинаково: их код заносится в словарь. Вновь образованное определение СМА функционально почти не отличается от прежнего определения через двоеточие. Единственное отличие состоит в том, что слово **C**, заносящее код операции в словарь, находится в части **DOES>** слова 1MI, а сам код (2F) — в поле параметров слова СМА.

Мы уже определили большую группу команд процессора 8080. Но остальные его команды не так просты. Например, команда ADD дополнительно вносит содержимое заданного регистра в регистр A (сумматор). Для того чтобы на обычном ассемблере 8080 добавить

содержимое регистра B к содержимому регистра A, нужно ввести

```
ADD B
```

Код операции ADD в двоичной системе имеет вид 10000SSS, где SSS — три бита, используемые для указания задаваемого регистра (S означает источник). Регистр B задается как 000, отсюда "ADD B" в двоичном коде будет выглядеть следующим образом: 10000000.

Аналогично если регистр L задается как 101 (S), то выражение "ADD L" превратится в 10000101. Иными словами, нужный код операции получается при выполнении команды OR над двоичным значением 10000000 (шестнадцатиричное 80) и числом, обозначающим регистр. Определим операцию ADD так:

```
: ADD  ( регистр# -- )   80  OR  C, ;
```

Самый простой способ занесения номера нужного регистра в вершину стека — задать номера регистров в виде констант:

```
0 CONSTANT B
1 CONSTANT C
2 CONSTANT D
3 CONSTANT E
4 CONSTANT H
5 CONSTANT L
7 CONSTANT A
```

(Здесь перечислены все регистры, которые можно использовать в команде ADD.) Теперь для занесения в словарь кода операции сложения значений регистра B и сумматора можно написать:

```
B ADD
```

Постфиксная запись ненамного усложняет дело, но зато ассемблер становится проще и сохраняется свойство расширяемости, присущее Форту (с помощью макроподстановки).

В некоторых командах, аналогичных ADD, значение регистра задается в трех младших битах. Поэтому имеет смысл для данного класса команд специфицировать свое определяющее слово:

```
: 2MI   CREATE C,   DOES>  ( регистр# -- ) C@ OR C, ;
```

С помощью этого слова можно ввести следующие определения:

```
80 2MI ADD     88 2MI ADC     90 2MI SUB     98 2MI SBB
A0 2MI ANA     A8 2MI XRA     B0 2MI ORA     B8 2MI CMP
```

(2MI функционирует аналогично 1MI, т. е. запоминает уникальный код операции определяемой команды (ребенка) в поле параметров последней. Отличие же заключается в том, что 2MI заставляет команду-ребенка при выполнении логически складывать посредством OR код операции ребенка с номером регистра из стека.)

Существует еще один класс машинных команд, содержащих номера регистров в коде операции, но в другом месте. Например, код команды INR (приращение) имеет вид 00DDD100 (шестнадцатиричное число 04), где DDD — регистр, подлежащий приращению (D означает «получатель»). Мы можем воспользоваться константами, обозначающими номера регистров, но при этом необходимо осуществить сдвиг на три бита влево, прежде чем команда OR сложит их с кодом операции (сдвиг на три позиции влево эквивалентен умножению на восемь):

```
: INR  ( регистр# -- ) 8 *  04 OR  C, ;
```

Как и в предыдущем случае, введем определяющее слово:

```
: 3MI   CREATE C,
   DOES> ( регистр# -- ) C@  SWAP 8 *  OR C, ;

04 3MI INR
```

Теперь выражение "C INR" занесет в словарь код операции: 00001100.

С помощью слова 3MI можно специфицировать еще один класс команд, в чем вы убедитесь, посмотрев листинг, приведенный в конце раздела.

Для создания команд остальных типов нам достаточно ввести всего два определяющих слова — 4MI и 5MI. Первое применяется для образования кодов тех операций, которые требуют дополнительно восьмиразрядного литерала, например ADI (непосредственное сложение с A). Второе слово определяет коды операций, требующих дополнительно 16-разрядного литерала. В качестве примера можно привести команды CALL, JMP и подобные им. Команды MOV, MVI и LXI уникальны и поэтому специфицируются индивидуально посредством двоеточия без использования определяющего слова.

Изучая листинг, обратите внимание на то, что в него включены операторы управления, такие, как **IF, ELSE, THEN, BEGIN, UNTIL, WHILE** и **REPEATE**. Это не совсем те слова, с определениями которых вы уже познакомились ранее (где передача управления компилируется посредством высокоуровневых слов Форта), а их версии, созданные только для ассемблера, где передача управления и разрешение адресов, как и в традиционном ассемблере, осу-

ществляются на уровне машинных команд. Однако они обеспечивают вам возможность программирования с использованием формата структур высокого уровня.

Но можно ли компилировать в словарь различные варианты слов **IF, THEN** и т. д., они ведь в нем смешаются? Конечно, так как команды ассемблера хранятся в контекстном словаре **ASSEMBLER**, а не в словаре **FORTH**. Определение **CODE** в нашем листинге инициирует слово **ASSEMBLER**, что делает этот контекстный словарь текущим всякий раз, когда мы начинаем CODE-определение.

Интересной особенностью Форт-ассемблера является и его расширяемость. Если в вашей программе имеются повторяющиеся фрагменты, то вы можете вместо них использовать *макрокоманды*. Ниже приводится пример макрокоманды, которая осуществляет «циклический сдвиг содержимого регистра А влево» и затем «добавляет содержимое регистра В»:

```
: SHIFT+   RLC  B ADD ;
```

Заметьте, что, появившись внутри ассемблерного определения, слово SHIFT+ помещает в словарь две команды, составляющие определение этого слова так, как если бы вместо него были введены сами команды:

```
RLC  B ADD
```

Адрес слова SHIFT+ не компилируется, а само оно при выполнении его кода не вызывается в качестве подпрограммы. Использование макросредств во время выполнения не приводит к каким-либо накладным расходам, поскольку машинные команды после макроподстановки в точности такие же, как и без нее.

Слово **NEXT** представляет собой одну из макрокоманд, написанных на языке Ассемблера. В нашей системе она определена так:

```
: NEXT   (NEXT) JMP ;
```

Иными словами, **NEXT** — это машинная команда, передающая управление по адресу, который оставляет в вершине стека слово (**NEXT**). По данному адресу расположен код адресного интерпретатора (о котором речь шла в гл. 9). Адресный интерпретатор является ядром Форта и выполняет поочередно все адреса в скомпилированном Форт-определении. Каждое определение через **CODE** должно заканчиваться инициированием адресного интерпретатора. Следовательно, любое ассемблерное определение должно завершаться словом **NEXT**. В нашем ассемблере определе-

ния также должны иметь в конце слово **END-CODE**, которое дополнительно восстанавливает контекст.

Ниже приводятся два примера, где используются команды описанного здесь ассемблера:

```
HEX
CODE ><    ( n -- n' )   \ Меняются местами старший и младший байты n
   H POP   L A MOV   H L MOV   A H MOV   H PUSH
   NEXT  END-CODE
```

(Мы пересылаем n из стека в пару регистров HL, регистр L (младшие байты) в регистр A, регистр H (старшие байты) в L, а A в H, помещаем содержимое пары регистров HL в стек, передаем управление **NEXT**).

```
CODE BP   ( a # -- )   \ Перевод из нижнего регистра в верхний
   D POP   H POP
   BEGIN   D A MOV   E ORA   0= NOT WHILE
      M A MOV  60 CPI   CS NOT   IF
         20 SUI   A M MOV   THEN
   D DCX   H INX   REPEAT   NEXT   END-CODE
```

(Пересылаем счетчик в пару регистров DE, а адрес — в пару регистров HL, начинаем цикл, проверяем, выполняя команду **OR** над содержимым регистров D и E, не равно ли значение счетчика нулю. Пока его значение не равно нулю, перемещаем символ из памяти, на которую ссылается указатель, в сумматор. Если код обрабатываемого символа больше 60 (строчная «а» и выше), вычитаем десятичное число 32, преобразуя символ в прописной, и записываем в память. Уменьшаем счетчик и увеличиваем адрес. Повторяем цикл. Передаем управление **NEXT**).

Преимущество работы на ассемблере такого вида заключается в том, что вы во время ассемблирования «находитесь в Форте». Если вам необходимо идентифицировать некоторое устройство с помощью имени, а не числа, вы можете определить его как обычную константу и присвоить ей имя внутри ассемблерного определения. Можно воспользоваться определением через двоеточие как макрокомандой или даже обратиться к переменной, поскольку она помещает в вершину стека свой адрес и поэтому может быть задействована в команде «непосредственной загрузки». Применение машинных команд раскрывает перед вами всю мощь языка Форт.

В основу описанного здесь ассемблера положен ассемблер 8080, разработанный Дж. Кассэди. Мы внесли в него изменения в соответствии со Стандартом-83 и для простоты изучения убрали некоторые зависимые от системы фрагменты. С оригиналом вы можете познакомиться в [1]. Ассемблер для других процессоров описан в [2], [3], [4].

```
\ Ассемблер 8Ø8Ø
\ учебная версия ассемблера 8Ø8Ø,основанная на фигФорте;
\ разработана Джоном Кэсседи
HEX
VOCABULARY ASSEMBLER
: CODE   CREATE  HERE HERE 2- !   ASSEMBLER ;
ASSEMBLER DEFINITIONS
: END-CODE   CURRENT @ CONTEXT ! ;
Ø CONSTANT B     1 CONSTANT C     2 CONSTANT D     3 CONSTANT E
4 CONSTANT H     5 CONSTANT L     6 CONSTANT PSW   6 CONSTANT M
6 CONSTANT SP    7 CONSTANT A
: 1MI   CREATE C,  DOES> C@ C, ;
: 2MI   CREATE C,  DOES> C@ OR C, ;
: 3MI   CREATE C,  DOES> C@  SWAP 8 * OR C, ;
: 4MI   CREATE C,  DOES> C@ C, C, ;
: 5MI   CREATE C,  DOES> C@ C, , ;

\ Ассемблер 8Ø8Ø
HEX
ØØ 1MI NOP      76 1MI HLT      F3 1MI DI       FB 1MI ED
Ø7 1MI RLC      ØF 1MI RRC      17 1MI RAL      1F 1MI RAR
E9 1MI PCHL     F9 1MI SPHL     E3 1MI XTHL     EB 1MI XCHG
27 1MI DAA      2F 1MI CMA      37 1MI STC      3F 1MI CMC
8Ø 2MI ADD      88 2MI ADC      9Ø 2MI SUB      98 2MI SBB
AØ 2MI ANA      A8 2MI XRA      BØ 2MI ORA      B8 2MI CMP
Ø9 3MI DAD      C1 3MI POP      C5 3MI PUSH     Ø2 3MI STAX
ØA 3MI LDAX     Ø4 3MI INR      Ø5 3MI DCR      Ø3 3MI INX
ØB 3MI DCX      C7 3MI RST      D3 4MI OUT      DB 4MI SBI
E6 4MI ANI      EE 4MI XRI      F6 4MI ORI      FE 4MI CPI
22 5MI SHLD     2A 5MI LHLD     32 5MI STA      3A 5MI LDA
CD 5MI CALL

\ Ассемблер 8Ø8Ø
HEX
C9 1MI RET        C3 5MI JMP        C2 CONSTANT Ø=   D2 CONSTANT CS
E2 CONSTANT PE   F2 CONSTANT Ø<
: NOT   8 OR ;
: MOV    8 *   4Ø + + C, ;
: MVI    8 *    6 + C, C, ;
: LXI    8 *    1+ C, , ;
: THEN   HERE SWAP ! ;
: IF   C,  HERE  Ø, ;
: ELSE   C3 IF SWAP THEN ;
: BEGIN   HERE ;
: UNTIL   C, , ;
: WHILE   IF ;
: REPEAT   SWAP  JMP  THEN ;
: NEXT   (NEXT) JMP ;
```

УСОВЕРШЕНСТВОВАННЫЙ ГЕНЕРАТОР БЕССМЫСЛЕННЫХ СООБЩЕНИЙ

Генератор бессмысленных сообщений, о котором шла речь в гл. 10, имеет существенный недостаток: необходимо осуществлять возврат каретки в определении слова СООБЩЕНИЕ. Это ведет к мозаичному выводу фраз.

Для того чтобы генерировать несколько сообщений, мы должны поручить самой программе фиксировать моменты, когда требуется возврат каретки. Следующий ниже листинг приводится без всяких

комментариев. Приобретайте навыки чтения программ на Форте. Попробуйте сделать определения более ясными. Результат работы нашей программы выглядит так:

```
В ДАННОМ СООБЩЕНИИ МЫ РАССКАЖЕМ ВАМ О ТОМ, ЧТО ВКЛАДЫВАЯ ИМЕЮЩИЕСЯ В НАЛИЧИИ
СРЕДСТВА В ИНТЕГРИРОВАННЫЙ ЦИФРОВОЙ КОМПЛЕКС ПРИМЕНЯЯ АВТОНОМНЫЙ КУЛЬТУРНЫЙ
ПРОДУКТ ПРЕДСТАВЛЯЕТСЯ ВОЗМОЖНЫМ ДАЖЕ НЕСМОТРЯ НА КВАЛИФИЦИРОВАННЫЙ ЦИФРОВОЙ
ПРОЕКТ ЕЩЕ БОЛЬШЕ КРЕПИТЬ УНИКАЛЬНЫЙ ПРОГРАММНЫЙ ОБЪЕМ.

С ОДНОЙ СТОРОНЫ, ИССЛЕДОВАНИЯ ПОКАЗАЛИ, ЧТО СТРУКТУРИРОВАННО ПРИМЕНЯЯ
ОБЩИЙ ТРУДОВОЙ ОБЪЕМ УЧИТЫВАЯ АВТОМАТИЗИРОВАННЫЙ ПРОГРАММНЫЙ АВТОМАТ
СТАНОВИТСЯ НЕОСУЩЕСТВИМЫМ ПОДНИМАЯ СЛУЧАЙНЫЙ ПРОИЗВОДСТВЕННЫЙ ПРОЦЕСС
ЕЩЕ БОЛЬШЕ КРЕПИТЬ КВАЛИФИЦИРОВАННЫЙ МНОГООТРАСЛЕВОЙ ПРИНЦИП.

С ДРУГОЙ СТОРОНЫ, ТЕМ НЕ МЕНЕЕ, ПРАКТИЧЕСКИЙ ОПЫТ ПОКАЗЫВАЕТ ,ЧТО
СТРУКТУРИРОВАННО ПРИМЕНЯЯ ЦИФРОВОЙ ХИМИЧЕСКИЙ КОМПЛЕКС УЧИТЫВАЯ НЕОБЫЧАЙНЫЙ
МНОГООТРАСЛЕВОЙ АВТОМАТ НЕОБХОДИМО РАССМАТРИВАЯ НЕОБЫЧАЙНЫЙ ПРОГРАММНЫЙ
КРИТЕРИЙ ФУНКЦИОНИРОВАТЬ КАК ВРЕМЕННЫЙ ЦИФРОВОЙ ПРОЕКТ.

В РЕЗУЛЬТАТЕ НАШЕ ПРЕДЛОЖЕНИЕ ЗАКЛЮЧАЕТСЯ В ТОМ, ЧТО СТРУКТУРНО ПРИМЕНЯЯ
СЛУЧАЙНЫЙ КУЛЬТУРНЫЙ ИНТЕРЕС УЧИТЫВАЯ СИСТЕМАТИЗИРОВАННЫЙ ЦИФРОВОЙ УРОВЕНЬ
НЕОБХОДИМО РАССМАТРИВАЯ ЦИФРОВОЙ ТРУДОВОЙ ПРОЕКТ ЕЩЕ БОЛЬШЕ КРЕПИТЬ
СИНХРОНИЗИРОВАННЫЙ КОРОТКОВОЛНОВЫЙ КРИТЕРИЙ.
```

```
Block# 156
( Генератор бессмысленных выражений, вариант с самоформатированием )
VARIABLE ОСТАТОК \ количество оставшихся символов, подлежащих сканированию
VARIABLE ПО-ГОРИЗОНТАЛИ \ текущее положение курсора для вывода по
   \ горизонтали
70 CONSTANT ПРАВГРАН \ правая граница
: СКАНЕР  ( длина-поиска -- адр-пробела ! конец-поля)
  2DUP +  ROT ROT  OVER + SWAP DO  I C@ BL =
     IF  DROP I LEAVE THEN  LOOP ;
: ДАЙСЛОВО  ( а -- а слово-счетчик) \ получ. очередн. слова для форматирования
   DUP  ОСТАТОК @  DUP  0> IF СКАНЕР ELSE DROP  THEN OVER -
   DUP  1+  NEGATE  ОСТАТОК +! ;
: СЮДА?  ( счетчик -- t=подходит для данной строки)
  ПО-ГОРИЗОНТАЛИ @ +  ПРАВГРАН < ;
: SPACE'   ПО-ГОРИЗОНТАЛИ @  IF SPACE  1 ПО-ГОРИЗОНТАЛИ +!  THEN ;
: CR'   CR  0 ПО-ГОРИЗОНТАЛИ ! ;
157 LOAD  158 LOAD

Block# 157
( Генератор бесмысленных выражений, вариант с самоформатированием )
: .СЛОВО  ( а # -- )  \ вывод слова, если необходимо, выполнение CR
  DUP  СЮДА? IF  SPACE'  ELSE CR'  THEN
  DUP  ПО-ГОРИЗОНТАЛИ +!   TYPE ;
: СЛЕДУЮЩЕЕ  ( а # -- след.адр.)
  +  1+ ;
: ВЫВОД  ( а #символов -- ) \ вывод сформатированного текста
  ОСТАТОК !
  BEGIN  ДАЙСЛОВО  DUP WHILE  2DUP .СЛОВО  СЛЕДУЮЩЕЕ  REPEAT
  2DROP ;
: АБРЕД  ( -- а)  161 BLOCK ;  \ случайные слова
: ОБОРОТЫ  ( -- а)  160 BLOCK ; \ связывающие обороты
: ВВЕДЕНИЯ  ( -- а)  159 BLOCK;\ начала фраз
: БРЕД  ( #строки #столбца -- а) \ получение адреса слова
  20 *  SWAP  64 * +  АБРЕД + ;

Block# 158
( Генератор бессмысленных выражений, вариант с самоформатированием )
: .БРЕД  ( #строки #столбца -- )  БРЕД  20 ВЫВОД ;
: ЧАСТЬ-РЕЧИ  ( #строки -- )  CREATE , \ определение частей речи
  DOES> @  10 CHOOSE  SWAP .БРЕД ;
0 ЧАСТЬ-РЕЧИ 1ПРИЛАГАТЕЛЬНОЕ
1 ЧАСТЬ-РЕЧИ 2ПРИЛАГАТЕЛЬНОЕ
2 ЧАСТЬ-РЕЧИ СУЩЕСТВИТЕЛЬНОЕ
: ФРАЗА   1ПРИЛАГАТЕЛЬНОЕ  2ПРИЛАГАТЕЛЬНОЕ  СУЩЕСТВИТЕЛЬНОЕ ;
: ОБОРОТ  ( #группы -- )  [ 4 64 * ] LITERAL   *
  3 CHOOSE  64 * +  ОБОРОТЫ +  64 ВЫВОД ;
: ПОВЕСТВОВАНИЕ   4 0 DO  I ОБОРОТ  ФРАЗА  LOOP  ." ."  CR' ;
: ВВЕДЕНИЕ  ( #абзаца -- )
  CR'  64 *  ВВЕДЕНИЯ +  64 ВЫВОД ;
: СООБЩЕНИЕ   CR'  CR'   4 0 DO  I ВВЕДЕНИЕ  ПОВЕСТВОВАНИЕ  LOOP ;
```

```
Block# 159
    В ДАННОМ СООБЩЕНИИ МЫ РАССКАЖЕМ ВАМ О ТОМ, ЧТО
    С ОДНОЙ СТОРОНЫ, ИССЛЕДОВАНИЯ ПОКАЗАЛИ, ЧТО
    С ДРУГОЙ СТОРОНЫ, ТЕМ НЕ МЕНЕЕ, ПРАКТИЧЕСКИЙ ОПЫТ ПОКАЗЫВАЕТ,ЧТО
    В РЕЗУЛЬТАТЕ НАШЕ ПРЕДЛОЖЕНИЕ ЗАКЛЮЧАЕТСЯ В ТОМ, ЧТО

Block# 160
    ПРИМЕНЯЯ
    ВКЛАДЫВАЯ ИМЕЮЩИЕСЯ В НАЛИЧИИ СРЕДСТВА В
    СТРУКТУРИРОВАННО ПРИМЕНЯЯ

    ИМЕЯ В ВИДУ
    ЧТОБЫ КОМПЕНСИРОВАТЬ
    УЧИТЫВАЯ

    ПРЕДСТАВЛЯЕТСЯ ВОЗМОЖНЫМ ДАЖЕ НЕСМОТРЯ НА
    СТАНОВИТСЯ НЕОСУЩЕСТВИМЫМ ПОДНИМАЯ
    НЕОБХОДИМО РАССМАТРИВАЯ

    ФУНКЦИОНИРОВАТЬ КАК
    СОЗДАТЬ
    ЕЩЕ БОЛЬШЕ КРЕПИТЬ

Block# 161
 ВЫСОКИЙ                  КУЛЬТУРНЫЙ          УРОВЕНЬ
 ОБЩИЙ                    ПРОИЗВОДСТВЕННЫЙ    ИНТЕРЕС
 АВТОМАТИЗИРОВАННЫЙ       НАУКОЕМКИЙ          КОМПЛЕКС
 ЗАПЛАНИРОВАННЫЙ          ВАЛОВОЙ             ОБЪЕМ
 ИНТЕГРИРОВАННЫЙ          ЦИФРОВОЙ            КОЭФФИЦИЕНТ
 КВАЛИФИЦИРОВАННЫЙ        МНОГООТРАСЛЕВОЙ     ПРИНЦИП
 ПРЕДСТАВИТЕЛЬНЫЙ         ХИМИЧЕСКИЙ          ГЕНЕРАТОР
 ТЕХНОЛОГИЧЕСКИЙ          НЕПРЕРЫВНЫЙ         ПРОЦЕСС
 АВТОНОМНЫЙ               АППАРАТНЫЙ          ИНТЕРФЕЙС
 ЦИФРОВОЙ                 НЕЗАВИСИМЬІЙ        АВТОМАТ
 СИНХРОНИЗИРОВАННЫЙ       ФУНКЦИОНАЛЬНЫЙ      КРИТЕРИЙ
 СИСТЕМАТИЗИРОВАННЫЙ      КОРОТКОВОЛНОВОЙ     ПРОЕКТ
 СЛУЧАЙНЫЙ                ОТРИЦАТЕЛЬНЫЙ       ИМПУЛЬС
 НЕОБЫЧАЙНЫЙ              ТРУДОВОЙ            ПОДЪЕМ
 ВРЕМЕННЫЙ                НЕХАРАКТЕРНЫЙ       СПАД
 УНИКАЛЬНЫЙ               ПРОГРАММНЫЙ         ПРОДУКТ
```

УПРАЖНЕНИЯ

12.1. Для описания четырех полей в простой файловой системе мы применяли следующие определения:

```
                          (смещение)         (длина)
CREATE фамилия                0 ,               16 ,
CREATE имя                   16 ,               12 ,
CREATE работа                28 ,               24 ,
CREATE телефон               52 ,               12 ,
```

а затем добавили еще одно

```
64 CONSTANT /ЗАПИСЬ
```

чтобы определить длину всей записи. Изменяя длину поля с фамилией, мы должны также подправлять начальный адрес остальных трех полей, а также значение переменной /ЗАПИСЬ. Найдите программный способ изменения этих значений. Определите синтаксис и напишите программу.

12.2. Используя язык базы данных, реализованной в простой файловой системе, определите новое слово «вызвать», которое осуществляло бы поиск имени и помещало в вершину стека ФИО и номер телефона, например:

```
Вызвать Конни
Конни Чанг  555-9653 ok
```

ЛИТЕРАТУРА

1. Cassady, John J., "8080 Assembler," *Forth Dimensions*, III/6, p. 180.
2. Duncan, Ray, "Forth 8086 Assembler," *Dr. Dobb's Journal*, 09/05, pp. 28–35 May 1984.
3. Perry, Michael A., "A 68000 FORTH Assembler," *Dr. Dobb's Journal*, 08/09, pp. 28–43, September 1983.
4. Ragsdale, William F., "A FORTH Assembler for the 6502," *Dr. Dobb's Journal* 06/09, pp. 12–24, September 1981; reprinted in *Forth Dimensions*, III/5, pp. 143 50, January/February 1982.

Приложение А. ОТВЕТЫ К УПРАЖНЕНИЯМ

ГЛАВА 1

```
1.    : ДАР   ." подставку для книг" ;
      : ДАРИТЕЛЬ   ." Маша" ;
      : БЛАГОДАРНОСТЬ   CR   ." Дорогая "   ДАРИТЕЛЬ   ." ,"   CR
         5 SPACES   ." спасибо за "   ДАР   ." . "   ;

2.    : МЕНЬШЕ-НА-ДЕСЯТЬ   ( n -- n-10 )   -10 + ;       или
      : МЕНЬШЕ-НА-ДЕСЯТЬ   ( n -- n-10 )   10 - ;
```

3. При компиляции определения БЛАГОДАРНОСТЬ компилятор включает в него определение слова ДАРИТЕЛЬ, существующее на момент компиляции. Если вы после компиляции слова БЛАГОДАРНОСТЬ добавите в словарь новый вариант слова ДАРИТЕЛЬ, то этот никак не отразится на уже скомпилированном слове БЛАГОДАРНОСТЬ. (Но вы можете переопределить (перекомпилировать) и слово БЛАГОДАРНОСТЬ. В этом случае в его определение войдет новый вариант слова ДАРИТЕЛЬ.)

ГЛАВА 2

```
1.    DUP DUP:    ( 1 2 -- 1 2 2 2)
      2DUP:       ( 1 2 -- 1 2 1 2)

2.    : NIP   ( a b -- b )   SWAP DROP ;

3.    : TUCK   ( a b -- b a b)   SWAP OVER ;

4.    : -ROT   ( a b c -- c a b)   ROT ROT ;
```

5. SWAP 2SWAP SWAP

6. : 3DUP (n1 n2 n3 -- n1 n2 n3 n1 n2 n3) DUP 2OVER ROT ;

7. : 2-7 (c a b -- n) OVER + * + ;

8. : 2-8 (a b -- n) 2DUP - ROT ROT + / ;

9. : УПАКОВКА (#яиц --)
   12 /MOD . ." коробок и " . ." не упаковано " ;

ГЛАВА 4

1.
```
-1 0= NOT . -1 ok
0 0= NOT . 0 ok
200 0= NOT . -1 ok
```

2. Не спросит ничего.

3. (употреблять спиртные напитки можно только с 21 года, тогда:)
```
: РАЗРЕШЕНИЕ ( возраст -- )
   20 > IF ." Употребление алкоголя разрешено "
   ELSE ." Вы еще молоды " THEN ;
```

4.
```
: ЗНАКИ ( n) DUP 0= IF ." Нуль " ELSE
             DUP 0< IF ." Отрицательное " ELSE
                       ." Положительное " THEN THEN DROP ;
```
(или как-нибудь иначе - лишь бы работало)

5. : <> (n1 n2 -- ?) = NOT ;

6.
```
: XOR ( x y -- ?)
   2DUP NOT AND SWAP ROT NOT AND OR ;
```

7. : STARS (n --) ?DUP IF STARS THEN ;

8.
```
: NEGATE ( n -- -n) 0 SWAP - ;
: ABS ( n -- !n! ) DUP 0< IF NEGATE THEN ;
```

9. : /UP (делимое делитель -- частное) /MOD SWAP IF 1+ THEN ;

10.
```
: -ROT ( a b c -- c a b) ROT ROT ;
: WITHIN ( n l h -- ?) -ROT OVER > NOT -ROT > AND ;
```
Ниже приводится более эффективный вариант, в котором используются приемы, рассмотренные в следующих главах:
```
: WITHIN ( n l h -- ?) OVER - >R - R> U< ;
```

11.
```
: УГАДАЙ ( ответ попытка -- ответ ) -- )
   2DUP = IF ." Вы угадали! " 2DROP ELSE
   2DUP < IF ." Слишком много " ELSE
             ." Слишком мало " THEN DROP THEN ;
```

12.
```
: .ОТРИЦАТЕЛЬНОЕ ( n -- !n!) 0< IF ." Отрицательное " ABS THEN ;
: ПРОПИСЬ ( n -- ) DUP ABS 4 > IF ." Выходит за границу " ELSE
   DUP .ОТРИЦАТЕЛЬНОЕ DUP 0= IF ." Нуль " ELSE
                      DUP 1 = IF ." Один " ELSE
                      DUP 2 = IF ." Два " ELSE
                      DUP 3 = IF ." Три " ELSE
                              ." Четыре "
   THEN THEN THEN THEN THEN DROP ;
```

13. в предположении, что -ROT и WITHIN уже загружены:
```
: 3DUP ( a b c -- a b c a b c) DUP 2OVER ROT ;
```

```
: ЛОВУШКА  ( ответ -меньш-число  -большее-число -- ответ ! -- )
  3DUP  OVER  =  -ROT = AND
     IF   ." Вы угадали! "  2DROP DROP  ELSE
  3DUP SWAP  1 + SWAP  WITHIN IF  ." Между "
     ELSE  ." Вне "  THEN  2DROP  THEN ;
```

ГЛАВА 5

1. -1 интерпретируется как число "отрицательная единица" ; 1- является словом Форта, которое вычитает единицу из значения на стеке.

2. `*/ NEGATE`

3. `MAX MAX MAX .`

4.
```
а) : 2ЗНАЧ  ( n1 n2 -- n? n?) \ помещение большего значения на вершину
      2DUP > IF SWAP THEN ;

б) : 3ЗНАЧ  ( n1 n2 n3 -- n? n? n?) \ большее значение на вершину
      2ЗНАЧ  >R  2ЗНАЧ R>  2ЗНАЧ ;
   ( Вы можете продолжать  в том же духе и далее ... )
   : 4ЗНАЧ \ из четырех элементов стека больший поместить в вершину
      3ЗНАЧ  >R  3ЗНАЧ  R>  2ЗНАЧ ;
   ...последним оператором во всех случаях должен быть 2ЗНАЧ.)

в) : ?ОБЪЕМ  ( длина ширина высота -- ) \ в любом порядке
      3ЗНАЧ  22 >  ROT 6 >  ROT 19 >
      AND AND  IF ." Подходит " THEN ;
                 ( Автор благодарит за пример Микаэла Хэма.)
```

5.
```
: РИСУЙ  ( n -- )  CR  80  100 */  STARS ;
```

6.
```
а)  0 32 -   10 18 */ . -17 ok
б)  212 32 -   10 18 */ . 100 ok
в)  -32 32 -   10 18 */ . -35 ok
г)  16 18 10 */  32 + . 60 ok
д)  233  273 - . - 40 ok
```

7.
```
: Ф>Ц  ( фаренг -- цельс)  32 -  10 18 */ ;
: Ц>Ф  ( цельс -- фаренг)  18 10 */  32 + ;
: К>Ц  ( кельв -- цельс)  273 - ;
: Ц>К  ( цельс -- кельв)  273 + ;
: Ф>К  ( фаренг -- кельв)  Ф>Ц  Ц>К ;
: К>Ф  ( кельв -- фаренг)  К>Ц  Ц>Ф ;
```

```
Block# 270
 \ Ответы к упражнениям; глава 6
 \ Упражнения 1 - 6
 : STARS  ( n)  0 ?DO  42 EMIT  LOOP ;
 : КЛЕТКА  ( ширина высота -- )  0 DO  CR  DUP STARS  LOOP  DROP ;
 : \STARS  ( #строк -- )  0 DO  CR  I SPACES  10 STARS  LOOP ;
 : /STARS  ( #строк -- )
     1-  0 SWAP  DO  CR  I SPACES  10 STARS  -1 +LOOP ;
 \ Определение /STARS с использованием конструкции BEGIN ... UNTIL:
 : A/STARS  ( #строк)
     BEGIN  1-  CR  DUP SPACES  10 STARS  DUP  0= UNTIL  DROP ;

 \ РОМБЫ определены в два этапа:
 : ТРЕУГОЛЬНИК  ( приращение граница индекс -- )
     DO  CR  9 I - SPACES  I 2* 1+ STARS  DUP +LOOP  DROP ;
 : РОМБЫ  ( #ромбов -- )
     0 DO  1 10 0 ТРЕУГОЛЬНИК  -1 0 9 ТРЕУГОЛЬНИК  LOOP  CR ;
```

```
Block# 271
 \ Ответы  к упражнениям; глава 6; продолжение
 \ Упражнение 7:
```

```
: THRU  ( от до -- )  1+ SWAP DO  I DUP .  LOAD  LOOP ;
\ Упражнение 8
: R%  ( n1 % -- n2)  10 */  5 +  10 / ;
: УДВОЕНО  ( вклад процент -- )
  OVER 2*  SWAP ROT  21 1 DO
     CR  ." Год "  I 2 .R   3 SPACES
        2DUP  R% +  DUP  ." Сумма "
     DUP  2OVER DROP > IF
     CR CR  ." Более чем удвоено через "  I .  ." лет "  LEAVE
  THEN LOOP  2DROP  DROP ;
\ Упражнение 9
: **  ( n1  n2 -- n1-в-степени-n2)
  1 SWAP  ?DUP IF  0 DO  OVER * LOOP THEN  SWAP DROP ;
      \ Спасибо за упражнение Дж.И.Андерсену, Эдинбург, Шотландия
```

Block# 272

```
\ Ответы к упражнениям; глава 7
\ Упражнение 1:
: N-MAX   0 BEGIN  1+  DUP 0< UNTIL 1- . ;
( Начиная с нуля, увеличиваем значение на стеке до тех пор, пока оно
  не станет отрицательным - это означает,  что достигнута граница пред-
  ставления целых чисел. Последний оператор 1- возвращает значение
  перед достижением границы.)

\ Упражнение 2 (а-е):
: BYNARY   2 BASE ! ;
: БИТОВЫЙ  ( #бита -- позиция-бита)  1 SWAP  0 ?DO  2*  LOOP ;
: УСТАНОВИТЬ-БИТ  ( битовый1 #бита -- битовый2)  БИТОВЫЙ OR ;
: ОЧИСТИТЬ-БИТ  ( битовый1 #бита -- битовый2)  БИТОВЫЙ -1 XOR  AND ;
: ДАЙ-БИТ  ( битовый #бита -- бит )  БИТОВЫЙ AND ;
: ПЕРЕКЛЮЧИТЬ-БИТ  ( битовый1 #бита -- битовый2)  БИТОВЫЙ XOR ;
: ИЗМЕНЕНИЕ  ( битовый1 битовый2 -- битовый3 )  XOR ;
```

Block# 273

```
\ Ответы к упражнениям; глава 7, продолжение
\ Упражнение 3:
: БИП   ." Бип "  7 EMIT ;
: ЗАДЕРЖКА   20000 0 DO LOOP ;
: 3ЗВОНКА   БИП ЗАДЕРЖКА БИП ЗАДЕРЖКА БИП ;

\ Упражнение 4а:
: Ф>Ц   -320 M+  10 18 M*/ ;
: Ц>Ф   18 10 M*/  320 M+ ;
: К>Ц   -2732 M+ ;
: Ц>К   2732 M+ ;
: Ф>К   Ф>Ц  Ц>К ;
: К>Ф   К>Ц  Ц>Ф ;
\ Упражнение 4б:
: .ГРАДУС  ( d -- )  DUP >R  DABS
   <#  # 46 HOLD  #S  R> SIGN  #>  TYPE SPACE ;
```

Block# 274

```
\ Ответы к упражнениям; глава 7, продолжение
\ Упражнение 5: ( в результате получается 17513;считается довольно долго.)
: ВЫЧИСЛ  ( x -- dv)
   DUP 7 M*  20 M+  ROT 1 M*/  5 M+ ;
: ?DMAX   0 BEGIN  1+ DUP  ВЫЧИСЛ  0 0 D< UNTIL  1- . ;

\ Упражнение 6:
В 16-чной системе десятичная цифра имеет такое же значение.

\ Упражнение 7:
: BINARY   2 BASE ! ;
```

```
: 3-СИСТЕМЫ
   17 0 DO  CR  ." Десятичная"   DECIMAL  I 4 .R  8 SPACES
                ." 16-ричная"    HEX      I 3 .R  8 SPACES
                ." Двоичная"     BINARY   I 8 .R  8 SPACES
   LOOP DECIMAL ;
```

Block# 275

```
\ Ответы к упражнениям; глава 7, продолжение
\ Упражнение 8:
( 3.7 интерпретируется как число двойной длины, поскольку содержит
  десятичную точку, и поэтому занимает два элемента стека.
  Так как 37 является небольшим числом, то его старшая часть
  состоит из нулей. "." является оператором над значением одинарной
  длины; две же точки подряд выводят обе части значения двойной
  длины. Старшая часть располагается на вершине стека, поэтому
  ее выводит первая точка: Вторая точка выводит младшую часть -
  37.)
( Число 65536 в точности на единицу превышает число, которое уме-
  щается в 16 разрядах. Поэтому 17-й разряд становится равным "1",
  а все остальные биты превращаются в нули. 17-й бит числа двойной
  длины является крайним правым битом старшей части. Она выводится
  как "1". Младшая часть выводится как нули.)
( Число 65538 больше на два, поэтому младшая часть выглядит как "2".)
```

Block# 276

```
\ Ответы к упражнениям; глава 7, продолжение
\ Упражнение 9:
( Поскольку данный фрагмент не является словом, форт интерпретирует
  его как число. Так как NUMBER интерпретирует точку как разделитель
  целой и дробной части числа, что для него является признаком числа
  двойной длины, то он на стек поместит ноль двойной длины.)

\ Упражнение 10:
: .ТЕЛЕФОН   ( d -- )   <#   # # # #   ASCII - HOLD   # # #
   OVER IF   ASCII / HOLD   #S THEN   #> TYPE SPACE ;
```

Block# 277

```
\ Ответы к упражнениям; глава 8
\ Упражнение 1-а
VARIABLE ПИРОЖКИ   0 ПИРОЖКИ !
: ИСПЕКИ-ПИРОЖОК    1 ПИРОЖКИ +! ;
: СъЕШЬ-ПИРОЖОК    ПИРОЖКИ @   IF -1 ПИРОЖКИ +!   ." Спасибо "
   ELSE  ." Какой пирожок? "   THEN ;
\ Упражнение 1-б
VARIABLE ЗАМОРОЖЕННЫЕ-ПИРОЖКИ    0 ЗАМОРОЖЕННЫЕ-ПИРОЖКИ !
: ЗАМОРОЗЬ-ПИРОЖКИ    ПИРОЖКИ @   ЗАМОРОЖЕННЫЕ-ПИРОЖКИ +!   0 ПИРОЖКИ ! ;
\ Упражнение 2:
: .БАЗА   BASE @   DUP DECIMAL .   BASE ! ;
\ Упражнение 3   ( сверх-надежный вариант):
: S>D   ( n -- d)   DUP   0< ;   \ из одинарной в двойную длину
: M.   ( d -- )   TUCK   DABS
   <#  DPL @  DUP  -1 <> IF   0 ?DO  #  LOOP   ASCII . HOLD ELSE
   DROP S>D   THEN  #S  ROT SIGN  #>  TYPE SPACE ;
```

Block# 278

```
\ Ответы к упражнениям; глава 8, Упражнение 4

CREATE #КАРАНДАШЕЙ   8 ALLOT   \ карандаши четырех цветов
0 CONSTANT КРАСНЫХ               2 CONSTANT ГОЛУБЫХ
4 CONSTANT ЗЕЛЕНЫХ               6 CONSTANT ОРАНЖЕВЫХ

: КАРАНДАШЕЙ   ( смещение -- а)   #КАРАНДАШЕЙ + ;

23 КРАСНЫХ КАРАНДАШЕЙ .
15 ГОЛУБЫХ КАРАНДАШЕЙ !
```

```
12 ЗЕЛЕНЫХ КАРАНДАШЕЙ !
 0 ОРАНЖЕВЫХ КАРАНДАШЕЙ !

\ Для проверки мы можем ввести, например, следующий текст:
\    ГОЛУБЫХ КАРАНДАШЕЙ ? 15 ok

Block# 279
\ Ответы к упражнениям; глава 8, Упражнение 5

CREATE 'ШАБЛОНЫ  20 ALLOT   ( 10 ячеек)
: ШАБЛОНЫ   ( i -- a ) 2*    'ШАБЛОНЫ + ;
: STARS   ?DUP IF   0 DO   42 EMIT  LOOP THEN ;
: ИНИЦ-ШАБЛОНОВ    10 0 DO   I 6 MOD   I ШАБЛОНЫ !   LOOP ;

: РИСУЙ   ( -- )
   10 0 DO   CR   I 2 .R   SPACE   I ШАБЛОНЫ @    STARS LOOP CR ;

ИНИЦ-ШАБЛОНОВ

Block# 280
\ Ответы к упражнениям; глава 8, упражнение 6
1 CONSTANT ЖЕНЩИНА                0 CONSTANT МУЖЧИНА
2 CONSTANT СЕМЕЙНЫЙ               0 CONSTANT ОДИНОКИЙ
4 CONSTANT РАБОТАЕТ               0 CONSTANT НЕ-РАБОТАЕТ
8 CONSTANT ГОРОДСКОЙ              0 CONSTANT НЕ-ГОРОДСКОЙ
VARIABLE ВАСЯ
VARIABLE ИРА
: ОПИСАНИЯ   ( состояние состояние состояние состояние
   >R   OR   OR   OR   R> ! ;
МУЖЧИНА СЕМЕЙНЫЙ НЕ-РАБОТАЕТ НЕ-ГОРОДСКОЙ  ВАСЯ ОПИСАНИЯ
ЖЕНЩИНА ОДИНОКИЙ РАБОТАЕТ ГОРОДСКОЙ  ИРА  ОПИСАНИЯ

Block# 281
\ Ответы к упражнениям; глава 8, упражнение 6, продолжение
: .ПОЛ   ( битовый -- )   женщина AND   IF ." жен" THEN   ." муж " ;
: .СЕМ-ПОЛ   ( битовый -- )
  СЕМЕЙНЫЙ   AND   IF ." семейный " ELSE ." одинокий " THEN ;
: .РАБОТА   ( битовый -- )
   РАБОТАЕТ AND   0=/F." не " THEN   ." работает " ;
: .ЖИТЕЛЬСТВО    ( битовый -- )
   ГОРОДСКОЙ AND   0=/F." не " THEN ." городской " ;
: СВЕДЕНИЯ   ( имярек -- )
   @ DUP .ПОЛ   DUP   .СЕМ-ПОЛ   DUP .РАБОТА   .ЖИТЕЛЬСТВО ;

Block# 282
\ Ответы к упражнениям; глава 8, упражнение 7
CREATE ПОЛЕ   9 ALLOT
: КВАДРАТ    ( #квадрата -- a)    ПОЛЕ + ;
: ОЧИСТИТЬ    ПОЛЕ    9 0 FILL ; ОЧИСТИТЬ
: ЛИНИЯ    ." | " ;
: ПОДЧЕРКИВАНИЕ    CR   9 0 DO   ASCII - EMIT LOOP CR ;
: .КЛЕТКА   ( #квадрата -- )   КВАДРАТ C@   DUP 0= IF   2 SPACES ELSE
   DUP 1 = IF   ." X "   ELSE   ." 0 "   THEN THEN DROP ;
: КАРТИНКА    CR   9 0 DO   I IF   I 3 MOD 0=   IF
```

```
    ПОДЧЕРКИВАНИЕ  ELSE  ЛИНИЯ  THEN THEN I .КЛЕТКА LOOP CR QUIT ;
: ХОД  ( игрок #квадрата -- )
   1-  0 MAX  8 MIN  КВАДРАТ C! ;
: X!  ( #квадрата -- )   1 SWAP  ХОД  КАРТИНКА ;
: 0!  ( #квадрата -- )  -1 SWAP  ХОД  КАРТИНКА ;
```

Block# 283

```
\ Ответы к упражнениям; глава 9

\ Упражнение 1:
VARIABLE  'ПОЛУЧАЕМ
: ПОЛУЧАЕМ  ( n n -- )    'ПОЛУЧАЕМ @ EXECUTE  . ;
: СКЛАДЫВАЯ   ['] +   'ПОЛУЧАЕМ ! ;
: УМНОЖАЯ   ['] *   'ПОЛУЧАЕМ ! ;

\ Упражнение 2:
\ Вы можете узнать это, введя
\    HERE U.
\ в начале работы или после применения системных команд, очи-
\ щающих словарь, таких как COLD или EMPTY.
```

Block# 284

```
\ Ответы к упражнениям; глава 9, продолжение

\ Упражнение 3:
\ Вы можете узнать это, введя
\     PAD HERE - U.

\ Упражнение 4:
\ а) Разницы нет. Переменная оставляет на стеке собственный pfa.
\ б) Пользовательская переменная оставляет на стеке адрес ячейки
     из пользовательской таблицы. Элемент словаря, который ищется
 словом, может находиться где угодно.
```

Block# 285

```
\ Ответы к упражнениям; глава 9, продолжение
\ Упражнение 5, Решение 1:
CREATE 'ЧТО-ДЕЛАТЬ  12 ALLOT   \ 6 ячеек
: ЧТО-ДЕЛАТЬ  ( i -- a)  0 MAX  5 MIN  2*  'ЧТО-ДЕЛАТЬ + ;

: ВСТРЕЧА   ." Привет, я говорю на Форте. " ;
: ПОСЛЕДОВАТЕЛЬНОСТЬ   11 1 DO  I .  LOOP ;
: ПЛИТКА   10 5 КЛЕТКА ;   \ См. ответы к главе 6
: НИЧЕГО ;

' ВСТРЕЧА    0 ЧТО-ДЕЛАТЬ !      ' ПОСЛЕДОВАТЕЛЬНОСТЬ  1 ЧТО-ДЕЛАТЬ !
' ПЛИТКА      2 ЧТО-ДЕЛАТЬ !      ' НИЧЕГО               3 ЧТО-ДЕЛАТЬ !
' НИЧЕГО     4 ЧТО-ДЕЛАТЬ !      ' НИЧЕГО               5 ЧТО-ДЕЛАТЬ !

: ЧТО-НИБУДЬ   ( индекс -- )  ЧТО-ДЕЛАТЬ @EXECUTE ;
```

Block# 286

```
\  Ответы к упражнениям; глава 9, продолжение
\ Упражнение 5, Решение 2:
CREATE 'ЧТО-ДЕЛАТЬ  12 ALLOT  \ 6 ячеек
: ЧТО-ДЕЛАТЬ  ( i -- a)  0 MAX  5 MIN  2*  'ЧТО-ДЕЛАТЬ +  ;

: ВСТРЕЧА   ." Привет, я говорю на  Форте. " ;
```

```
: ПОСЛЕДОВАТЕЛЬНОСТЬ   11 1 DO  I .  LOOP ;
: ПЛИТКА   10 5 КЛЕТКА ;    \ см. ответы к главе 6
: НИЧЕГО ;

: ИНИЦИАЛИЗАЦИЯ    ( -- )
   6 0 DO  ['] НИЧЕГО  I  ЧТО-ДЕЛАТЬ !  LOOP
   ['] ВСТРЕЧА   0 ЧТО-ДЕЛАТЬ !  ['] ПОСЛЕДОВАТЕЛЬНОСТЬ  1 ЧТО-ДЕЛАТЬ !
   ['] ПЛИТКА        2 ЧТО-ДЕЛАТЬ ! ;
ИНИЦИАЛИЗАЦИЯ
: ЧТО-НИБУДЬ  ( индекс -- )  ЧТО-ДЕЛАТЬ  @EXECUTE ;
```

Block# 287

```
\ Ответы к упражнениям; глава 10
\ Упражнение 1:
: СИМВОЛ    ( i -- a)
   228 BLOCK + ;
: ЗАМЕНА  ( c1 c2 -- )   \ замена c1 на c2
   1024 0 DO  OVER  I СИМВОЛ C@   = IF  DUP  I СИМВОЛ C!
   UPDATE  THEN  LOOP  2DROP ;

\ Упражнение 2:
181 LOAD \ Случайные числа
\ ??? CONSTANT ПРЕДСКАЗАНИЯ \ номер блока под сообщения
: ПРЕДСКАЗАНИЕ  CR  16 CHOOSE  64 *  ПРЕДСКАЗАНИЯ BLOCK +
   64 -TRAILING TYPE  SPACE ;
\ Вы можете обращаться к своим собственным "предсказаниям".Занесите
\ их по одному на строку в свободный блок, а затем занесите номер
\ этого блока в приведенное выше определение константы ПРЕДСКАЗАНИЯ
```

Block# 288

```
\ Ответы к упражнениям; глава 10, продолжение
\ часть а:
: ДА/НЕТ?  ( -- t=Y | f=прочее)  KEY  DUP EMIT  ASCII Y = ;
\ часть б:
: ДА/НЕТ?  ( -- t=Y | f=прочее)
   KEY  95 AND  DUP EMIT  ASCII Y = ;
\ часть в; два возможных решения:
: ДА/НЕТ?  ( -- t=Y | f=N )
   BEGIN KEY  95 AND  DUP  ASCII Y = IF  DROP TRUE EXIT  ELSE
                      DUP  ASCII N = IF  0= EXIT  THEN THEN
  DROP  FALSE UNTIL ;

: ДА/НЕТ?  ( -- t=Y | f=N )
   BEGIN  KEY  95 AND  DUP ASCII Y =  OVER ASCII N =  OR NOT
      WHILE DROP REPEAT  ASCII Y = ;

```

Block# 289

```
\ Ответы к упражнениям; глава 10, упражнение 4                1
: ЖИВОТНЫЕ  LIT" КРЫСЫ  БЫКА   ТИГРА  КРОЛИКА ДРАКОНА ЗМЕИ   ЛОШАДИ
 БАРАНА  ОБЕЗЬЯНЫПЕТУХА  СОБАКИ  СВИНЬИ  " ;
: .ЖИВОТНОЕ  ( u -- )  \  u изменяется от 0 до 11
   6 *  ЖИВОТНЫЕ 1+  +  6 -TRAILING TYPE ;
: (ГОРОСКОП)  ( год -- )
   1900 -  12 MOD
   ." Вы родились в год "  .ЖИВОТНОЕ
   ASCII . EMIT  CR ;
350 351 THRU  \ загрузка определения EXPECT#
: ЦИФРЫ  ( #цифр -- d )
   DUP 0 DO  ASCII   EMIT  LOOP  DUP 0 DO  ЗАБОЙ  LOOP
   PAD  SWAP  2DUP 1+ BLANK  EXPECT# DROP  PAD 1-  NUMBER ;
: ГОРОСКОП
   CR  ." В каком году вы родились? "  4 ЦИФРЫ
   CR  DROP (ГОРОСКОП) ;
```

Block# 290

```
\ Ответы к упражнениям; глава 10, упражнение 5
VARIABLE СТРОКА
: начало   0 строка ! ;
```

```
: добавить \  1прилагательное, 2прилагательное, 3прилагательное,
           \  существительное  ( -- )
   СТРОКА @ 0 БРЕД  60 BLANK UPDATE
   3 0 DO
      ASCII , WORD COUNT  СТРОКА @  I БРЕД SWAP CMOVE  UPDATE
   LOOP  1 СТРОКА +! ;
\ или, используя TEXT:
: добавить \  1прилагательное, 2прилагательное, 3прилагательное,
           \  существительное  ( -- )
   3 0 DO
      ASCII , TEXT  PAD  СТРОКА @ I БРЕД  20 CMOVE  UPDATE
   LOOP  1 СТРОКА +! ;
Block# 291
\ Ответы к упражнениям; глава 10
\ Упражнение 6
: >ДАТА     ( a -- n n )
    0 0 ROT CONVERT  ROT >R  0 SWAP CONVERT  ROT >R
    0 SWAP  CONVERT  2DROP  1900 +  R>  R> 256 * +  SWAP ;
: СКАНИРОВАНИЕ   BL WORD >ДАТА ;

\ Упражнение 7
VARIABLE  STUFF   \ первым блоком файла
300 STUEF \ является блок 300
: ЭЛЕМЕНТ  ( i -- a)
    2*  1024 /MOD  STUFF @  + BLOCK + UPDATE ;
\ Проверка виртуального массива:
: ИНИЦИАЛ-МАССИВ   600 0 DO  I  I ЭЛЕМЕНТ !  LOOP  ;
: .МАССИВ   600 0 DO  I . SPACE  I ЭЛЕМЕНТ ?  LOOP ;
Block# 292
\ Ответы к упражнениям; глава 10, упражнение 7, продолжение

\ Теперь преобразуем виртуальный массив в файл:
: ИСПОЛЬЗОВАНО  ( -- a)  СВОБ @  BLOCK  UPDATE ;
\ Переопределим ЭЛЕМЕНТ так, чтобы ИСПОЛЬЗОВАННЫЕ пропускались:
: ЭЛЕМЕНТ  ( i -- a)
   1+  2* 1024 /MOD  СВОБ @  + BLOCK  +   UPDATE ;

: НЕТ-ИСП   0 ИСПОЛЬЗОВАНО ! ;
НЕТ-ИСП
: ПОМЕСТИТЬ  ( n -- ) ИСПОЛЬЗОВАНО @ ЭЛЕМЕНТ ! 1 ИСПОЛЬЗОВАНО +! ;
: ВНЕСТИ  ( n1 n2 -- )  SWAP  ПОМЕСТИТЬ ПОМЕСТИТЬ ;

: ТАБЛИЦА  CR  ИСПОЛЬЗОВАНО @  0 ?DO  I 8 MOD  0= IF  CR  THEN
   I ЭЛЕМЕНТ @  8 .R LOOP  CR ;
Block# 293
\ Ответы к упражнениям; глава 11
\ Упражнение 1:
: ЗАГРУЗКА  ( n -- )  CREATE ,  DOES>  ( -- )  @ LOAD ;

\ Упражнение 2:
: ОСНОВАНИЕ.  ( n -- )  CREATE ,
   DOES>  ( n -- )  @  BASE @ SWAP  BASE !  SWAP .  BASE ! ;

\ Упражнение 3:
: МНОГО   ( a -- )  CREATE ,
   DOES>  ( -- )  @  SWAP  0 ?DO  DUP  EXECUTE LOOP  DROP ;
' CR МНОГО CRS
4 CRS
```

```
Block# 294
\ Ответы к упражнениям; глава 11, продолжение
\ Упражнение 4:
: TURNE   [COMPILE] DO ; IMMEDIATE
: RETURNE  [COMPILE] LOOP  ; IMMEDIATE
: ПОПЫТКА   10 0 TURNE  I . RETURNE ;

\ Упражнение 5:
: ЦИКЛЫ  ( #раз -- )
   >IN @  SWAP 0 DO  DUP >IN !  INTERPRET  LOOP DROP ;
```

```
Block# 295
\ Ответы к упражнениям; глава 11, упражнение 6:
: STAR  42 EMIT ;
: .РЯД  ( b -- )  \ вывод звездочки на каждый бит из байта
   CR  8 0 DO  DUP 128 AND  IF STAR ELSE SPACE THEN
     2* LOOP  DROP ;
VARIABLE ШАБЛОН
: БИТ  ( t=не-пробел -- )
   1 AND  ШАБЛОН @ 2*  +  ШАБЛОН !  ;
: ЗВЕЗДЫ>БИТЫ  ( a -- b)
   0 ШАБЛОН !  8 OVER + SWAP DO  I C@  BL <> БИТ  LOOP
   ШАБЛОН @ ;
: ЧТ-РЯД  ( -- b)
   ASCII ! WORD  COUNT +  8 -  ЗВЕЗДЫ>БИТЫ ;
: ФОРМА  CREATE  8 0 DO ЧТ-РЯД C, . LOOP
   DOES>  8 OVER + SWAP DO I C@ .РЯД  LOOP  CR ;
```

```
Block# 296
ФОРМА .L  XXX     |
           X      |
           X      |
           X      |
           X      |
           X      |
           X     X|
          XXXXXXXX|
ФОРМА .B  XXXXXXX |
           X     X|
           X     X|
           XXXXXX |
           X     X|
           X     X|
           X     X|
          XXXXXXX |
```

```
Block# 297
\ Ответы к упражнениям; глава 12, упражнение 1:
: ширина  ( смещение длина -- нов-смещение )
   CREATE  OVER ,  DUP ,  + ;
0  \ начальная позиция внутри записи
16 ширина фамилия
12 ширина имя
24 ширина работа
12 ширина телефон
CONSTANT /ЗАПИСЬ   ( количество байт на запись ) EXIT
По приведенным выше правилам компилируются одинаковые структуры.
"ширина" является определяющим словом, которое хранит на стеке
текущую позицию внутри поля. Для каждого поля "ширина" компили-
рует смещение и длину, а затем их складывает, чтобы получить
следующее смещение. После того, как определится "телефон", по-
лученное в результате значение является длиной всей записи и
превращается в константу /ЗАПИСЬ
```

```
Block# 298                                                        Block# 299
\ Ответы к упражнениям; глава 12, упражнение 2:
: вызвать  ( имя  ( -- )
  имя запомнить  ВВЕРХ -НАЙТИ IF  ОТСУТСТВУЕТ
     ELSE  CR .ИМЯ  телефон .ПОЛЕ  THEN ;
```

Приложение Б

АЛФАВИТ СЛОВ ФОРТА

В данном перечне слова расположены в алфавитном порядке согласно коду ASCII. Здесь проставлены номера страниц, где приводится формальное описание слова. Числа в скобках относятся к сноскам.

Приложение В

СЛОВА ФОРТА, СГРУППИРОВАННЫЕ ПО ТЕМАМ

Здесь указаны страницы, содержащие формальное описание слов. ЧИсла в скобках относятся к сноскам.

АРИФМЕТИКА

ОДИНАРНАЯ ДЛИНА

УПРАВЛЕНИЕ СТРУКТУРОЙ

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА

ВИРТУАЛЬНАЯ ПАМЯТЬ

ОГЛАВЛЕНИЕ

Научное издание

Броуди Лео

НАЧАЛЬНЫЙ КУРС ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ФОРТ

Книга одобрена на заседании секции редсовета
по электронной обработке данных в экономике 28.10.86

Зав. редакцией К. В. Коробов

Редактор Н. К. Логинова. Мл. редакторы Т. Т. Гришкова, Н. Е. Мендрова. Худож. редактор Ю. И. Артюхов. Техн. редактор И. В. Юдинцева. Корректоры Т. М. Васильева, Н. П. Сперанская. Переплет художника Б. С. Вехтера.

ИБ № 2264

Сдано в набор 6.06.89. Подписано в печать 01.12.89. Формат $60 \times 88^1/_{16}$. Бум. кн.-журнальная. Гарнитура Литературная. Печать офсетная. Усл. п. л. 21,56. Усл. кр.-отт. 21,56. Уч.-изд. л. 20,08. Тираж 40 000 экз. Заказ 676. Цена 1 р. 70 к.
Издательство «Финансы и статистика», 101000, Москва, ул. Чернышевского, 7.

Типография им. Котлякова издательства «Финансы и статистика» Государственного комитета СССР по печати. 195273, Ленинград, ул. Руставели, 13.